

FILE COPY

RADC-TR-88-324, Vol VIII (of nine)
Interim Report
March 1989



AD-A209 453

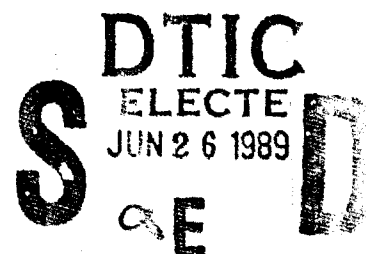
**NORTHEAST ARTIFICIAL
INTELLIGENCE CONSORTIUM ANNUAL
REPORT 1987 Knowledge Base
Maintenance Using Logic Programming
Methodologies**

Syracuse University

Kenneth A. Bowen

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

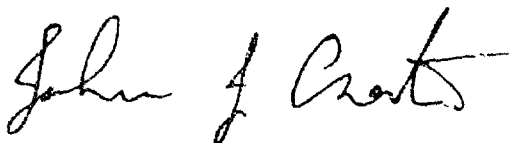


89 6 23 030

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

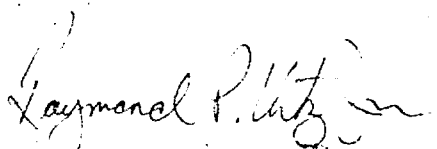
RADC-TR-88-324, Vol VIII (of nine) has been reviewed and is approved for publication.

APPROVED:



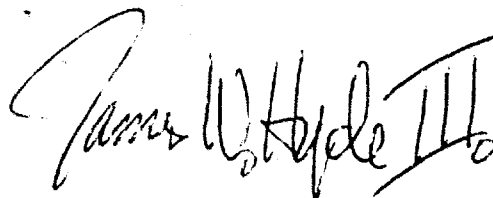
JOHN J. CROWTER
Project Engineer

APPROVED:



RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER:



JAMES W. HYDE, III
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COES) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188		
1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS N/A			
2a. SECURITY CLASSIFICATION AUTHORITY N/A			3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.			
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A						
4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A			5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-88-324, Vol VIII (of nine)			
6a. NAME OF PERFORMING ORGANIZATION Northeast Artificial Intelligence Consortium (NAIC)		6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COES)			
6c. ADDRESS (City, State, and ZIP Code) 409 Link Hall Syracuse University Syracuse NY 13244-1240			7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700			
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center		8b. OFFICE SYMBOL (If applicable) COES	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-85-C-0008			
6c. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700			10. SOURCE OF FUNDING NUMBERS			
			PROGRAM ELEMENT NO. 61102F 62702F	PROJECT NO. 2304 5581	TASK NO. J5 27	WORK UNIT ACCESSION NO. 01 13
11. TITLE (Include Security Classification) NORTHEAST ARTIFICIAL INTELLIGENCE CONSORTIUM ANNUAL REPORT 1987 Knowledge Base Maintenance Using Logic Programming Methodologies						
12. PERSONAL AUTHOR(S) Kenneth A. Bowen						
13a. TYPE OF REPORT Interim		13b. TIME COVERED FROM Dec 86 TO Dec 87		14. DATE OF REPORT (Year, Month, Day) March 1989		
15. PAGE COUNT 52						
16. SUPPLEMENTARY NOTATION N/A						
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)			
FIELD	GROUP	SUB-GROUP	Artificial Intelligence, Prolog, Knowledge Base, Logic Programming, Languages, Feasibility, Knowledge Base, Maintenance.			
12	05					
12	07					
19. ABSTRACT (Continue on reverse if necessary and identify by block number) The Northeast Artificial Intelligence Consortium (NAIC) was created by the Air Force Systems Command, Rome Air Development Center, and the Office of Scientific Research. Its purpose is to conduct pertinent research in artificial intelligence and to perform activities ancillary to this research. This report describes progress that has been made in the third year of the existence of the NAIC on the technical research tasks undertaken at the member universi- ties. The topics covered in general are: versatile expert system for equipment maintenance, distributed AI for communications system control, automatic photo interpretation, time- oriented problem solving, speech understanding systems, knowledge base maintenance, hardware architectures for very large systems, knowledge-based reasoning and planning, and a knowledge acquisition, assistance, and explanation system. The specific topic for this volume is the use of logic programming methodologies for knowledge base maintenance.						
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED			
22a. NAME OF RESPONSIBLE INDIVIDUAL JOHN J. CROWTER			22b. TELEPHONE (Include Area Code) (315) 330-3577		22c. OFFICE SYMBOL RADC/COES	

DD Form 1473, JUN 86

Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

NAIC
Northeast Artificial Intelligence Consortium
1987 Annual Report

Volume 8

Inference Techniques for
Knowledge Base Maintenance
using
Logic Programming Methodologies

Prof. Kenneth A. Bowen, PI

Staff:

Hamid Bacha, Aida Batarekh, Assoc. Prof. Howard Blair, Ilyas Cicekli,
Keith Hughes, Visiting Asst. Prof. Hyung-Sik Park, V.S. Subrahmanian

Logic Programming Research Group
School of Computer and Information Science
313 Link Hall, Syracuse University
Syracuse, NY 13210

Table of Contents

8.1 Executive Summary	8-2
8.2 Introduction	8-3
8.3 Hamid Bacha	8-7
<i>metaProlog Implementation and Application</i>	
8.4 Aida Batarekh	8-9
<i>Incomplete and Inconsistent Knowledge</i>	
8.5 Howard Blair and V.S. Subrahmanian	8-11
<i>Theory of Logic Programming</i>	
8.6 Kenneth Bowen	8-16
<i>Foundations and Application: Reason Maintenance</i>	
8.7 Ilyas Cicekli	8-34
<i>metaProlog Implementation</i>	
8.8 Keith Hughes	8-37
<i>Interfaces to Databases</i>	
8.9 Hyung-Sik Park	8-43
<i>Negation and Databases</i>	

8.1 Executive Summary

During the past year, work focused about three primary endeavors: completing implementation of the metaProlog compiler, development of interfaces between Prolog systems and traditional relational databases, and exploration of theoretical issues concerning logic and knowledge bases. Considerable success was achieved in all three areas:

- We have completed implementation of two variant compilers for the metaProlog language, the difference lying in their approach to the maintenance of relationships between theories (i.e., logic databases of clauses). Both are based on the same underlying byte-code interpreting abstract machine, with the compiler proper coded in C. Both are interactive, incremental compilers supporting last-call optimization, garbage collection, and interactive debugging of compiled code. In addition, both are approximately as fast as our original compiler for Prolog, running at approximately 10,000 LIPS on the naive reverse benchmark on a Sun 3/280. Experience with Prolog implementation indicates that a production-grade native code compiler for metaProlog can be expected to achieve a 20-fold speed-up over the present compilers. We have begun experimenting with these compilers, building knowledge base maintenance programs and expert systems in metaProlog.
- We have developed general tools for constructing quite direct interfaces between our first Prolog compiler and traditional relational databases (in this case, ORACLE and INGRES). These interfaces make the tuples in an external database appear to ordinary Prolog programs as if they (the set of tuples) were a large collection of Prolog facts. The interfaces are two-way: not only can tuples in the database be transparently read from Prolog, but asserts on the corresponding predicate(s) in Prolog cause tuples to be written into the database. The techniques used in constructing these interfaces will be applicable to any external relational database which supports interfaces to foreign C-coded programs.
- We have considerably extended our understanding of a number of theoretical issues concerning the mathematical foundations of Prolog, metaProlog, logic-based default reasoning, and the interaction between negation and deduction from databases. These will be detailed below.

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	



8.2 Introduction

8.2.1. Logic and Databases: The Need to Extend Prolog

Prolog has many attractive features as a programming tool for artificial intelligence and the management of knowledge bases. These include code that is easy to understand, programs that are easy to modify, and a clear relation between its logical and procedural semantics. Moreover, it has proved possible to create clear and efficient implementations. Nonetheless, it possesses several shortcomings. Chief among these is difficulty representing dynamic databases (databases which change in time) and an apparent restriction to backward chaining, backtracking, depth-first search. A major component of our work has been to develop and implement an extension to Prolog, called *metaProlog*, which preserves the virtues of Prolog while introducing powerful constructions to attack these problems. This work is a direct continuation of the investigation into meta-level programming in logic begun by Bowen and Kowalski [1982].

Many artificial intelligence applications demand facilities which amount to the ability to dynamically manipulate databases or knowledge bases. A database is most naturally represented in Prolog as a set of assertions and clauses. This exploits all the advantages of Prolog's inherent deductive machinery. However, the logical core of ordinary Prolog provides no conceptual basis for segmenting or modifying the database. Most implementations of Prolog have provided ad hoc extensions to the basic logic programming paradigm which allow for dynamic modification of the program database by the program itself. But since the database is the program, the use of these facilities introduces difficulties similar to those introduced by global variables and self-modifying code in conventional programming languages. The effect of these features on the virtues listed above is catastrophic. Programs become difficult to understand, reliable modification of the code is almost impossible, and the logical semantics is utterly destroyed. We know of no mathematical or philosophical definition of first-order proof where the collection of axioms is not fixed. We would suspect any such notion to be incoherent. We believe these difficulties can be overcome by the introduction of theories as first-class objects which can be dynamically created and passed as parameters. In standard Prolog, goals are invoked with respect to a single background theory. In *metaProlog*, goals must be proved in an explicitly identified theory. We regard this system as simply a first-order logical theory of axiom sets and proofs.

The means of indicating that a *metaProlog* goal *G* should be solved in a particular theory *T* is an explicit call on the proof predicate *demo*. From a logical point of view, the proof predicate is really a relation between three objects: the *theory T*, the *goal G*, and the *proof P* which attests to the solvability of *G* in *T*. But logic programming is not only concerned with the static existence of proofs, but also the process of discovering them. That is, it is also concerned with the notion of search space and search strategies. Thus, for logic programming, the deep central relation is the one which holds between a theory *T*, a goal *G*, and the complex object consisting of a proof-for *G* in *T* seen as a portion of a search space explored by a particular search strategy. Our investigations have led us to the conclusion that all of these entities must be treated as first-class objects (*metaProlog* terms) capable of being manipulated and passed as values of parameters.

8.2.2. Meta-Level Programming

It is important to make clear our notion of meta-level programming. Briefly, one distinguishes between the formal language being used to conduct some (unspecified) axiomatic investigation (the object language) and the language used to carry on any discussion about the object language (the metalanguage). For many purposes (including those of this paper), the metalanguage need only be powerful enough to discuss the combinatorial syntactic properties of the object language. The essential point is that the relations of the metalanguage are about the syntactic entities of the object language: the variables of the metalanguage range over various syntactic entities of the object language. In contrast, the variables of the object language either have no specified range

(when it is viewed as a formally uninterpreted language) or (when the object language is treated as being interpreted). range over the members (possibly extremely mathematically complex) of some specified set.

Properly viewed, an ordinary Prolog interpreter is already a meta-level object. The object level consists of a fragment of ordinary first-order logic, a language and proof predicate. The latter describes which formulas of the language are consequences of sets of other formulas of the language. The meta-level of a theorem-prover is concerned with the manipulation of sets of object-level formulas in the search for a collection of formulas which witnesses the derivability of a given goal formula from a given set of axiom formulas. The prover proper is a meta-level object because its variables range over formulas (and other syntactic classes) of the object level language.

Thus a Prolog interpreter really defines a relationship between sets of formulas (the program database), goal formulas, and proofs, namely the relation that the proof witnesses the deducibility of the goal formula from the program database. (Note that the standard Prolog interpreters return a portion of the proof to the user, namely that part of the substitution applying to the variables occurring in the goal). As commonly implemented, pure Prolog interpreters incorporate the program database as a fixed part of the interpreter. Thus, from a meta-level point of view, a standard Prolog interpreter provided with a fixed program database defines a certain meta-level unary predicate applying to goal formulas. This meta-level unary predicate holds for just those goal formulas which are deducible from the program database by the interpreter. The fundamental operator of standard Prolog systems is thus a one-place operator (usually written `call(...)`) which invokes a search for a deduction of its argument from the implicit program database parameter. The heart of the proposal set forth by Bowen and Kowalski was to utilize a system implementing the full deducibility relation described above. Such a system would have metavariables which not only range over formulas and terms, but would also allow the metavariables to range over sets of formulas (called theories). The fundamental operator of such a system is a three-place operator, usually written `demo(Theory,Goal,Proof)`, which invokes a search for a proof of the goal formula appearing as its second argument from the theory (or program) appearing as its first argument.

All metaProlog program databases are the values of metaProlog variables and are set up either by reading them in from files or by dynamically constructing them using system predicates. Besides the built-in predicate `demo/3`, the system predicates include:

- `add_to(Theory, Axiom, NewTheory)`
- `drop_from(Theory, Axiom, NewTheory)`

which build new theories from old ones by adding or deleting formulas. Thus for example, one might find the body of a clause containing calls of the form

(*) ..., `add_to(T1, A, T2)`, `demo(T2, D, P)`,...

where the theory which is the value of `T1` has been constructed by the earlier calls. The effect of (*) would then be to construct a new theory `T2` resulting from `T1` by the addition of the formula `A` as a new axiom, and then the invocation of a search for a proof of the formula `D` from the theory `T2`. Since `demo` implements the proof relation, such programs as (*) preserve the logical semantics of Prolog while providing for the dynamic construction of new databases from old.

The correctness and completeness of an implementation of `demo` are expressed by what were called reflection rules by Bowen and Kowalski:

- If `demo(T, A, P)`, then `A` is derivable from `T` via proof `P`.
- If `A` is derivable from `T` via proof `P`, then `demo(T, A, P)`.

8.2.3. The General Situation

The real power (meta_power) of this system lies not in the specific system facilities we have described, but in the programming methodology they introduce. The example in the preceeding section only beings to explore the possibilities of this system. Using this approach, in [Bowen and Kowalski 1982], [Bowen and Weinberg 1985], and [Bowen 1985] we have begun to logically characterize frames and default hierarchies, generalized networks of theories and semantic nets, and more general control strategies such as bottom-up or breadth-first search. There is no logical requirement that the only notion of proof in metaProlog be the Horn clause-oriented demo predicate we have introduced. We see no reason why other methods of proof cannot co-exist with demo. We envisage the situation in which another method of proof would be rapidly prototyped using explicit recursive calls on the present demo, and later integrated into the system at a low level.

By stepping up to the full meta_level point of view wherein all components of the system have become first-class objects, we have entered the realm of a logical construal of Theories, Goals, and SearchSpaces in which it is possible to axiomatically and programmatically characterize elements of the system previously regarded as parts of the implementation. This allows us to introduce powerful logical approaches to the construction of artificial intelligence systems, and in particular, to systems which must manipulate complex knowledge bases.

8.2.4. Current Status and Accomplishments

Last year we began evolving a compiler for metaProlog from a compiler for Prolog which we had constructed earlier. This work involved considerable modification of the underlying abstract Prolog machine which is the target of the compiler proper. As we neared the final stages of installing the high-level machinery for manipulation of theories proper, we discovered two variant approaches to the maintenance of relationship among theories. Since we could see advantages to both, we simply finished off (during this year) by completing two variant compilers for metaProlog. Both support the original Prolog syntax as well as the metaProlog syntax, and both are based on the same underlying abstract machine and compiler front-end. Like our earlier Prolog compiler, both are incremental, interactive systems supporting last-call optimization, garbage collection (one of the important achievements of this year's work) and interactive debugging of compiled code. The underlying abstract machine is a byte-code interpreter. The system is approximately as efficient as our earlier Prolog: approximately 10,000 LIPS for the naive reverse benchmark running on a Sun 3/280. This makes it one of the fastest non-commercial Prolog-type systems. Experience with the construction of high-speed native code compilers for ordinary Prolog indicates that a native code compiler for metaProlog would run at speeds approaching 20 times the present system(s). Speeds such as these provide a solid basis for implementation of quite substantial AI systems.

Observation of many AI systems (both Prolog- and non-Prolog-based) shows that quite a few of them utilize a moderately large number of complex rules coupled with very large numbers of basic facts. Such large sets of facts put considerable strain on the virtual paging of standard operating systems. For this reason, it is important to develop methods of storing these large sets of facts in databases external to the Prolog or metaProlog systems proper. While extraordinarily large sets of facts may require specialized hardware, there are compelling reasons for exploring the use of existing relational database management systems (RDBMS) for this purpose:

- Collections of Prolog facts are logically isomorphic to the tuples of RDBMS tables;
- For many potential AI systems, the necessary sets of facts already exist in RDBMS tables; moreover, these tuples must remain accessible to existing non-AI applications while at the same time, the AI application must remain "up-to-date" with the RDBMS as affected by the existing applications;
- If efficient interfaces can be constructed, considerable effort would be saved through the utilization of the facilities of existing RDBMS, including locking, audit trailing, etc.

During the course of this year, we clearly demonstrated that quite efficient interfaces can indeed be constructed. The interfaces are completely transparent to the Prolog programmer: the tuples in the RDBMS tables simply appear as if they really made up a collection of Prolog facts. Not only can they be accessed in read mode by Prolog programs, but the Prolog program (using the ordinary Prolog assert builtin) can write tuples back into the database, and (using the Prolog retract) it can remove tuple from the database. Interfaces were built between our earlier Prolog and ORACLE and INGRES databases. On the one hand, these interfaces can be extended to metaProlog, while on the other hand, the techniques used in constructing them can be applied to any RDBMS which supports interfaces to foreign C-coded programs.

These concrete systems developments provide extremely promising programming environments for the construction of sophisticated and large AI programs. Consequently, it is quite important to extend our understanding of the foundations of Prolog and metaProlog, as well as the more general relationships between logic, negation, default reasoning, and databases. As we neared the end of the prototyping work described above, we intensified our work on these theoretical issues, partly by re-directing the efforts of some staff members, and partly by adding new staff this year. Since the theoretical questions pursued are rather varied, they are described in detail in the sections below (as is the prototyping outline above).

8.2.5. References

Bowen, K.A., *Meta-Level Programming and Knowledge Representation*, *New Generation Computing* 3 (1985) pp. 359-383.

Bowen, K.A. and Kowalski, R.A., *Amalgamating language and metalanguage in logic programming*, in *Logic Programming*, eds. Clark and Tarnlund, Academic Press, 1982, pp. 153-172.

Bowen, K.A. and Weinberg, T., *A Meta-Level Extension of Prolog*, in *Proceedings of the 1985 Symposium on Logic Programming*, J. Cohen and J. Conery, eds, IEEE Press, Washington, D.C., 1985, pp. 48-53.

8.3 Hamid Bacha

metaProlog Implementation and Application

The work carried out during this period proceeded along two tracks which consisted of the design and implementation of the logic-based language MetaProlog and a (Meta)Prolog-based medical expert system in the area of Acid-Base and Electrolytes disorders. Though diverse as may seem, the work in the two areas is complementary. On the one hand, we would like to develop a powerful language using the most advanced methods available to achieve high speed execution and provide a suitable programming environment capable of supporting sophisticated large-scale applications. On the other hand, we would like to test these capabilities by developing a large-scale application such as a medical expert system that tries to combine both clinical and patho-physiological knowledge in its reasoning in order to arrive at a better diagnosis. We will briefly summarize the work accomplished so far starting with MetaProlog where a compiler, a debugger, and a user manual have been completed, then talk about the prototype medical expert system implemented in Prolog and in the process of being enhanced and redone in the more expressive language MetaProlog.

8.3.1 MetaProlog Compiler

We have designed and implemented a fast and efficient MetaProlog compiler. It is a byte-code interpreter based on an extended Warren Abstract Machine instructions set. It subsumes the full Prolog language as well as its syntax. It is a fast incremental compiler which looks and feels just like an interpreter. It was designed to run under the UNIX operating system, but was since ported to VAX/VMS. The main feature is the treatment of theories as first class objects, thus allowing the existence and manipulation of multiple databases (contexts) simultaneously. Having endowed the theories with the status of first-class objects, the provability relation of the object language is easily axiomatized. The two-place predicate 'demo' names the provability relation of the object language and is defined in terms of sentences of the meta-language. However, because of the amalgamation of the object language and the meta-language, object-level and meta-level formulas are freely intermixed. The results from one level are readily available to the other level because the correct representability of the provability relation guarantees that simulation in the metalanguage is equivalent to and interchangeable with direct execution in the object language [Kowalski79]. This idea is captured by what was called linking rules in [Bowen82]. To solve $\text{demo}(T', G')$ in MetaProlog, we only need to switch to the context of the theory named by T' and invoke the MetaProlog machinery to show $T \vdash G$, that is G is derivable from the theory T . From a programming point of view, demo can be seen as a way of temporarily switching context to a new theory. After the goal specified by demo is proved, context is switched back to the old theory. Context switching is achieved either implicitly through the demo predicate, or explicitly using the setcontext predicate. Proofs are also treated as first-class objects. A three argument version of 'demo' is used when the proof that witnesses the derivation of a goal from a theory is to be returned as a value of a variable. Future versions of MetaProlog will allow a partial proof to be specified in advance. This partial proof will serve to guide the search along a predetermined path in some portions of the search space. We allow many theories to be viewed as one big theory and used to prove a certain goal. However, since it is very inefficient to combine all those theories into a large one, especially if it is never going to be used again, we introduce the concept of a virtual theory. A virtual theory is the conjunction of many theories which are viewed as one large theory without the expense of actually building the new theory. Except for a small overhead (creation of an extra choice point), using the virtual theory is as fast as using the large theory. If we take into consideration the overhead associated with building the large theory, using the virtual theory is by far more efficient.

8.3.2 MetaProlog Debugger

A MetaProlog debugger has been written fully in MetaProlog. It has all the features of a standard four port Prolog debugger. In addition, it shows the theory in which the call is being evaluated. It

does not use any assert or retract during debugging. This is a very useful feature since assert/retract create a lot of garbage. Since MetaProlog puts the code on the heap, that would tend to fill it up very quickly.

8.3.3 MetaProlog User Manual

We wrote a comprehensive user manual to accompany the MetaProlog system. It highlights the main features of the language and provides some examples. It also includes a summary of all MetaProlog commands. However, this manual is restricted to the MetaProlog features that are not part of the Prolog language. A suitable user manual for Prolog is suggested to complement the MetaProlog one. Our Logic Programming Research Group has one such manual for Columbus Prolog which was designed and implemented by the same group.

8.3.4 (Meta)Prolog-Based Medical Expert System

A prototype medical expert system in the area of Acid-Base and Electrolytes disorders has been implemented in the Prolog language. It is an attempt to smoothly integrate reasoning at both the clinical level (or surface level reasoning) and the pathophysiological level (or deep reasoning). The diagnostic methodology adopted uses meta-level knowledge to restrict the diseases considered to a very small number. As the meta-level restrictions are progressively lifted, the number of diseases to be considered is increased. One way of achieving this is by focusing on a specific area of the body (e.g. lungs) considered the most likely given the initial findings and the laboratory data. As we fail to come up with a diagnosis, other areas (e.g. heart, kidney, ...) are progressively added as meta-level restrictions are lifted. This methodology is opposite to that used by many medical expert systems which start with a large number of diseases and progressively eliminates most of them until a diagnosis is reached. Some of the advantages are that a diagnosis is reached very quickly most of the time (since most cases are usually routine), the diagnosis is very focused, and the hard cases are considered only when necessary. This prototype expert system is being rewritten in MetaProlog to take advantage of its superior expressive power. One of the advantages of MetaProlog is the use of theories to pursue several hypotheses at the same time. The hypotheses can then be reevaluated whenever necessary to decide on the next course of action. Since the hypotheses are represented by separate theories, they can be compared to determine which one accounts for most of the findings and symptoms. They can also be combined to study whether the conjunction of multiple diseases can account for some of the findings that are not explained by any single hypothesis alone.

8.3.5 Publications

Meta-level Programming: A Compiled Approach, Proceedings of the Fourth International Conference on Logic Programming, Edited by Jean-Louis Lassez, MIT Press, 1987.

MetaProlog Design and Implementation, In Preparation.

8.3.6 References

[Kowalski79]: Kowalski, R.A., *Logic for Problem Solving*, Elsevier North Holland Inc, New York, 1979.

[Bowen82] Bowen, K.A. and Kowalski, R.A., *Amalgamating Language and Metalanguage*, in *Logic Programming*, eds. Clark K.L. and Tarnlund S.A., Academic Press, New York, pp. 153-172.

8.4 Aida Batarek:

Incomplete and Inconsistent Knowledge

This research, which will ultimately be part of a dissertation currently under preparation, focuses on problems and potential remedies in reasoning in the presence of incomplete and/or inconsistent knowledge. One of the goals of this research is the implementation of a reasoning agent capable of dealing with such knowledge. As the closed world assumption no longer holds in such a setting, the rule of inference known as negation by failure is inappropriate for deriving negative facts. Hence, a different type of negation is needed, as well as a larger set of inference rules – one that allows proofs by contradiction. Furthermore, the agent should be able to deal with hypothetical reasoning by advancing plausible assumptions, when appropriate, and studying their consequences. Assumptions which turn out to be inconsistent with known information have to be dropped from further consideration in the solution of the problem at hand.

8.4.1 The ∇ operator for hypothetical reasoning

The operator ∇ is used to introduce assumptions and reason hypothetically. Combined with literals, it forms a ∇ -literal (or p -literal). A ∇ -literal has one of the following forms: ∇L (a p^+ -literal) or $\neg\nabla L$ (a p^- -literal) where L is a literal. A *cube* is a literal or a ∇ -literal. A general ∇ -clause has the form

$$H \leftarrow B_1 \& \dots \& B_n$$

where each of H, B_1, \dots, B_n is a cube and $n \geq 0$.

Unlike most work on non-monotonic reasoning, where the possibility operator is allowed only in the body ([RR80], [MD80], [DE87]), a p -literal can appear in both the head or the body of a clause. This gives more expressive power and allows the distinction between definite and indefinite knowledge. Logical consequences which depend on no assumptions are considered *definite*, whereas *indefinite* consequences depend on at least one assumption. When a p^+ -literal appears in the head of a clause, it allows the *conditional* introduction of an assumption, the conditions being that the body of the clause be solvable as a goal. When it appears in the body, it allows an unconditional introduction of an assumption.

8.4.2 Consistent Clusters of an Inconsistent ∇ -Logic Program

When inconsistent information is present in a ∇ -logic program, no models exist and no soundness or completeness results are applicable. Yet, it generally is the case that the inconsistent information is localized and does not affect all the knowledge available from the program. The program can be divided in clusters, or families, in such a way that families are independent of each other, and one can study the set of models of the consistent families. A method for clusterizing the program is presented for the propositional case and it is shown that there is a least model of the consistent families. A proof procedure is also given for which soundness and completeness results exist with respect to that least model. A generalization of the above method and proof procedure is currently under investigation.

Also being researched, is a way of preventing proofs based on contradictory information from introducing new knowledge. As a contradictory piece of evidence is both true and false, its truth (or falsity) may be used as support in the derivation of the truth (or falsity) of some fact. It is our point of view that such an evidence such not be used in any proof until its status has been resolved.

8.4.3 ∇ -Thinker

The ∇ -Thinker is the current Prolog Implementation of some of the results developed so far. When presented with a ∇ -Logic program and a goal G to solve, the system will provide a proof if one

can be found and reply **yes**, or provide a counterproof (ie, a proof for $\neg G$) and reply **no**. The proofs can be definite or indefinite, depending on whether some assumptions had to be made or not in order to find a proof. A definite proof is always searched for first, and assumptions introduced if none can be found and the user has used the ∇ operator. If neither a proof nor a counterproof can be found, the system replies **unknown**. The above system is being expanded to respond **contradictory** when a goal is both true and false (definite) and \star when it is both true and false (indefinite). The above responses will correspond to the truth values in the lattice $\mathcal{L} = \{t, f, dt, df, k, u, \star\}$ used by Ginsberg ([MG86]) for non-monotonic logic, and which is an expansion of Belnap's four-valued lattice ([NB75]).

8.4.4 References

- [NB75] Belnap N, *A useful Four-Valued Logic*, in : Modern Uses of Multiple-Valued Logic, Dunn and Epstein, 1975.
- [DE87] Etherington D, *Formalizing NonMonotonic Reasoning Systems*, AI (1987) 31 : 41-85.
- [MG86] Ginsberg M, *Multiple-Valued Logic :I*, AAAI 86.
- [MD80] McDermott D and Doyle J, *Non-Monotonic Logic I*, AI 13 : 41-72.
- [RR80] Reiter R, *A Logic for Default Reasoning*, AI (1980) 13 : 81-132.

8.5 Howard A. Blair, V. S. Subrahmanian:

Theory of Logic Programming

8.5.1 Quantitative Logic Programming

Van Emden's formulation of quantitative rule sets [VE86] does not allow negated literals to appear in the body of any rule. Furthermore, naively extending Van Emden's formalism by permitting the occurrence of negated atoms in the bodies of rules causes computational difficulties. In order to address these issues, Subrahmanian proposed in [Su87a] the formalism of quantitative logic programs (QLPs). A technical device called a truth value annotation is introduced, and it is shown that such annotations are powerful enough to replace negation. A declarative semantics is given for QLPs, and it is shown that QLPs have a least model that coincides with the least fixed point of a monotone operator mapping interpretations to interpretations. Furthermore, an operational semantics is given for QLPs based on Van Emden's and/or tree searching method. In addition, strong completeness results have recently been obtained [Su87b], and it is shown that the greatest supported model of such QLPs is semi-computable. This has applications in reasoning about beliefs. In [Su87c], an extension of [Su87a] is proposed which allows annotations to be a pair of reals in the $[0,1]$ interval; the first component of the pair expresses the degree of belief in a proposition, while the second expresses the degree of disbelief in that proposition. It is shown that this method also allows a simple characterization of the degree of inconsistency (or over-definedness) of any atom.

In earlier work, we attempted to give methods for computing the greatest supported model of a class of logic programs, viz. the well-behaved generally Horn programs. This method has since been extended to apply to quantitative logic programs. Completeness results have been obtained for a natural subclass of programs. It has also been found that the theoretical framework for quantitative logic programming can be extended to the case where the set of truth values (annotations) can be any complete lattice (subject to a restriction on the definition of negation). [Su87a], mentioned above, also introduced the notion of quantitative logic programs, exploring their declarative semantics. The operational semantics given in [Su87a] was extended significantly during this period - in particular, the notion of correct answer substitution was introduced and soundness and completeness results obtained. In addition, the completeness results for the and-or tree searching technique given in [Su87a] were strengthened to be applicable to quantitative logic programs that are not well covered, thus removing one restriction in the completeness theorem obtained in [Su87a]. In addition, the soundness and completeness results for SLDq-resolution in [Su87a] were strengthened to apply to any *nice* (a technical term here) QLP. Moreover, these latter soundness and completeness results are applicable to existential queries unlike the results of [Su87a] and [VE86] which are applicable to ground queries only.

8.5.2 Inconsistency in Logic Programming

Our work in this area may roughly be classified into two parts; in the first part, [BS87a,BS87b], we have attempted to enhance the expressive power of pure logic programming by allowing clauses of the form $L_0 \leftarrow L_1 \& \dots \& L_k$ where each L_i is a literal. Since such programs can be inconsistent, it is necessary to give a formal semantics for programs that may express inconsistency. A formal semantics based on a 4-valued Belnap lattice is given, and a non-classical notion of model is introduced. It is shown that such programs have a least model which is identical to the least fixed point of a monotone operator, and that this least fixed-point is semi-computable. A specific class of programs, viz. the well-behaved programs, is introduced, and these programs are shown to possess certain interesting model-theoretic properties (as well as being computationally more tractable). A sound and complete operational semantics is given, and for well-behaved programs, an SLD-resolution-like proof procedure is described. [BS87b] is a substantial extension of [BS87a]: a new operational

semantics, together with much more powerful completeness results are given, a new class of queries called *hypothetical queries* is introduced and a mechanism to process such queries is described; computational methods of computing the greatest supported models of a class of programs are given, and a canonicity theorem is proved.

In the second part, we have attempted to define a notion of entailment called CS-entailment that is defined in terms of classical entailment. It is shown that an atom is true in the least model (in the sense of Fitting's 3-valued semantics) [Fi85] of the Fitting-completion of a program iff it is CS-entailed by the completion (but not Fitting's completion!) of the program. This allows the development of very large logic programs whose completions may be globally inconsistent, but locally consistent. (cf. [BS87c]). In addition, it is shown that using this technique, hypothetical queries can be soundly answered.

We prepared and submitted a research proposal to the National Science Foundation, [Bl87a]. The proposal concerns the study of the foundations of knowledge bases modeled by general logic programs based on paraconsistent and related nonclassical logics. We use the term *paraconsistent* generically to refer to any formal system which allows sound reasoning with inconsistent information sources. Previously, we developed an abstract notion of logic programs and their model theory over a large variety of nonclassical, multi-valued logics. We discuss this below in the section on *Abstract Semantics*. Our abstract development includes a variety of schemes for paraconsistent knowledge bases in support of research involving several interrelated themes: computational reasoning in the presence of inconsistency and indefiniteness, and reasoning with *hypotheses* that may be inconsistent with a given knowledge base formulated as a logic program are two. This entails formally describing logic programs based on locally consistent, globally inconsistent logics, and identifying computationally tractable fragments of logics of this kind, analogous to the Horn clause fragment of first-order logic.

8.5.3 Abstract Semantics

[Su87a], [BS87a], [BLS87], [VE86], [Fi85] and [MP85] all investigate the semantics of logic programs that naturally arise from variations of classical logic. The motivation for this departure from classical logic is centered on the need for a rigorous semantics for treating approaches to evidential and nonmonotonic reasoning in the presence of uncertainty. [BBS87] takes an abstract approach that unifies the semantics of these previously studied, semantically differing kinds of logic programs into a single theoretical framework within which to specify their semantics. This illuminates the underlying algebraic reasons for the striking similarity seen in each of these earlier studies regarding the model theory of the programs that were under investigation. Moreover the work also gives a model-theoretic semantics for logics of belief-maintenance systems.

Current work on the abstract semantics of logic programs is centered on constructing a semantics for Metalogic programs. Truth values in the underlying logic can consist of logic programs themselves. The naturally induced ordering on the programs corresponds to the subset relation on the ground versions of the programs. This is felt to be too fine grained a point of view upon which to construct a semantics, so we are now attempting to find a suitable quotient of this ordering, as well studying the space of possible quotients to determine the intrinsic ordering among programs for the purposes of characterizing the intentional component of them. An ordering which renders programs equivalent only when they are logically equivalent is too course-grained.

8.5.4 Protected Circumscription

Minker and Perlis, in a 1985 paper [MP85], have suggested that one might wish to prevent the use of the negation as failure rule with respect to certain atoms. Their investigation applies to function-free pure logic programs. We extend this proposal to general logic programs, and characterize the

models of protected completions of such programs in terms of the fixed points of a certain operator. This proposal applies to completely arbitrary general logic programs. In addition, an operational semantics is given that is proved to be sound and complete for a large class of general logic programs.

8.5.5 Elimination of Local Variables

Much, but not all, of the model theory (semantics) for logic programs, whether based on two-valued classical logic, or multi-valued or even continuous valued nonclassical logic suitable for evidential nonmonotonic reasoning, is symmetric with respect to positive and negative relations. A substantial part of the asymmetry which does arise in the theory is due to the presence of so-called internal variables in procedure bodies analogous to the internal variables in procedure body definitions in Algol-like languages. In [Bl87c] we give a transform for producing from a given program, a semantically related program without internal variables, and specify, in terms of the standard semantics for quantifiers, the logical relationship between the given program and its transform.

8.5.6 Stratification-independent Semantics for Stratified Deductive Databases

The semantics of stratified logic programs was developed in [ABW87] and exploited to achieve good heuristic means for performing updates in stratified deductive databases in [AP87]. In essence a *deductive database* (DDB) consists of relation tables together with function-free deductive rules represented as general definite clauses (i.e. clauses with or without occurrences of negation.) The DDB is *stratified* when there is no recursive dependency between relations and their negations as in

$$\begin{array}{lcl} R_1 & \leftarrow & R_2 \ \& \ \neg R_3 \\ R_3 & \leftarrow & R_1 \end{array}$$

In [Bl87e] we show how to determine the extension of the DDB by means that is independent of the stratification. The significance of this lies in the fact that the DDB's rules usually have multiple *stratifications*. While it was known that the declarative meaning of the DDB was independent of the stratification, it was unknown how to naturally define the declarative meaning of the DDB in a natural way, declaratively itself, that was independent of the stratification. [Bl87e] eliminates this deficiency in the theory. One paper was completed and another is in preparation. Specifically, in [AB87], co-authored with K. R. Apt of the University of Texas at Austin, the perfect models of stratified programs were shown to have arbitrarily high arithmetic complexity in a way depending on the number of strata in the program. (Stratified programs were studied by Apt, Blair and Walker, [ABW87].) This entails that instances of deductive databases whose rules make unrestricted use of function symbols are highly noncomputable. [AB87] classifies this noncomputability with precision. Blair has shown that if a stratified program is free of internal variables, then the complexity of the program's perfect model can be held to zero-jump, independently of the number of strata in the program. The paper detailing these results is in preparation, [Bl87b]. This encourages the idea that reasonable disciplines can be found that allow the use of function symbols but still determine computable perfect models.

8.5.7 Canonical Conservative Extensions

This work is also concerned with achieving symmetry between positive and negative relations in the semantics of logic programs. Specifically, given a logic program of definite clauses which may possess certain asymmetry between the semantics of success and finite failure, [Bl87f] shows how to effectively determine a semantically equivalent program with the asymmetry removed. The results

improve upon very similar results reported in [JS86] by eliminating the expansion of the underlying domain, called the Herbrand universe of the program. [Bl87d] makes a further improvement on these results by distinguishing between so-called conservative extensions in the classical sense to programs and conservative approximations that are not extensions but still include sufficient power to answer all of the queries that could be answered by the original program.

References

- [AB87] Apt, K. R. and Blair, H. A. "Arithmetic Classification of Perfect Models of Stratified Programs". Draft. November, 1987.
- [ABW87] Apt, K. A., Blair, H. A., and Walker, A. "Towards a Theory of Declarative Knowledge". in *Foundations of Deductive Databases and Logic Programming*, Jack Minker, ed. Morgan-Kaufmann, 1987.
- [AP87] Apt, K. & Pugin, J.-M. "Maintenance of Stratified Databases Viewed as a Belief Revision System". *Proceedings: Principles of Database Systems*, 1987.
- [BBS87] Blair, H. A., Brown, A. L. and Subrahmanian, V. S. "A Model-Oriented Foundation for Belief Maintenance". (Working draft prepared.)
- [Bl87a] Blair, H. A. *Computational Reasoning with Paraconsistent and Nonclassical Logics*. Research Proposal submitted to the National Science Foundation.
- [Bl87b] Blair, H. A. "Bounding the Arithmetic Complexity of Perfect Models of Stratified Programs". (In preparation.)
- [Bl87c] Blair, H. A. "Elimination of Local Variables". (Working draft prepared.)
- [Bl87d] Blair, H. A. "Canonical Conservative Approximations of Logic Programs." (In preparation.)
- [Bl87e] Blair, H. A. "Stratification-independent Inductively Defined Semantics of Stratified Logic Programs". (in preparation)
- [Bl87f] Blair, H. A. "Canonical Conservative Extensions of Logic Program Completions" IEEE Symposium on Logic Programming, San Francisco, August, 1987.
- [BLS87] Blair, H. A., Lu, J. J., Subrahmanian, V. S. "A Kripke-Kleene Semantics for Protected Completions of General Logic Programs", to be submitted. (Working draft prepared.)
- [BS87a] Blair, H. A., Subrahmanian, V. S. "Paraconsistent Logic Programming", *Proc. 7th Foundations of Software Technology & Theoretical Computer Science Conference*, Lecture Notes in Computer Science, Springer Verlag, Dec. 1987.
- [BS87b] Blair, H. A., Subrahmanian, V. S. "Foundations of Generally-Horn Logic Programming", in preparation. (Working draft prepared.)
- [BS87c] Blair, H. A., Subrahmanian, V. S. "General Logic Programs with Locally Consistent Completions", to be submitted. (Working draft prepared.)
- [Fi85] Fitting, M. "A Kripke-Kleene Semantics for Logic Programs". *J. of Logic Programming*, 1985, no. 4, pp. 295-312.

- [JS86] Jaffar, J. & Stuckey, P. "Canonical Logic Programs". *J. of Logic Programming*, vol. 3, no. 2. pp. 143-155. 1986.
- [MP85] Minker, J. & Perlis, D. "Computing Protected Circumscription". *J. of Logic Programming*, vol. 2, no. 4. pps 235-249. 1985
- [Su87a] Subrahmanian, V. S. "On the Semantics of Quantitative Logic Programs" , em Proc. 4th IEEE Symp. on Logic Programming, San Francisco, Sept., 1987.
- [Su87b] Subrahmanian, V. S. "Query Processing in Quantitative Logic Programming", to be submitted.
- [Su87c] Subrahmanian, V. S. "Towards a Theory of Evidential Reasoning in Logic Programming", *Logic Colloquium '87*.
- [VE86] Van Emden, M. "Quantitative Deduction and Its Fixpoint Theory", *Journal of Logic Programming*, vol. 3, no. 1, pp. 37-53. 1986.

8.6 Kenneth A. Bowen:

Foundations and Application: Reason Maintenance

8.6.1 The Logic of Monotonic Reasoning

8.6.1.1 Introduction

An intelligent artifact (program) reasoning about some aspect of the world must often maintain a collection assertions representing its current beliefs. In many settings, these assertions may only be plausible conjectures which may have to be retracted depending upon the course of the agent's reasoning and the accumulation of evidence. In such circumstances, the agent's reasoning is said to be *non-monotonic*. If the assertions added to the collection of beliefs are never retracted, the agent's reasoning is said to be *monotonic* [refs].

The course of development of the set of beliefs is dependent both on the (external) evidence discovered and on the agent's choices of reasoning steps to employ, even in the monotonic case. This picture of monotonic reasoning is strikingly similar to the intuitionistic descriptions of the idealized mathematician [ref-Brouwer] and its formalization in the *theory of constructions* [refs]. While the treatment of infinite totalities in intuitionism is a fascinating and problematic topic, a strict point of view can maintain that such totalities are only *potential* and never *actual*. This position is founded on the view that the idealized mathematician is a finite being acting in time, and that all actualized mathematical entities must be constructed by the mathematician. Consequently, not only must all actual entities be in fact finite, but the totality of constructed entities and verified assertions is necessarily finite at any point in time.

This strict finiteness of the collection of entities and verified assertions is certainly characteristic of the belief sets of intelligent computer programs. But like the intuitionist mathematician's sets, these sets are *potentially infinite* in that there is no in principle bound on the effort of either the ideal mathematician or the intelligent program.

Kripke's introduction of a classical model theory of intuitionistic logic provided a powerful tool for the classical analysis of intuitionistic reasoning. This model theory is also very attractive for the analysis of the reasoning of intelligent artifacts (in the non-monotonic case, in its *modal* incarnation). While the introduction of Kripke models provided a intuitively appealing set-theoretic interpretation of intuitionistic statements, these models have the disadvantage that in dealing with arithmetic and analysis, or for that matter, any theory in which all of the statements "there exist of least n individuals" are derivable, the interpretation requires that infinitely many individuals actually exist at each world-point or situation, thus preventing a direct interpretation of any concepts of potentially infinite totality. This drawback also applies to the analysis of the reasoning of artificial agents. In this paper, we provide a modification of Kripke's approach which allows us to restrict the number of individuals actually existing at any world-point or situation to be finite. The price we pay is that the number of world-points is necessarily infinite, and in the interpretation of the logical operators, we must universally quantify over subcollections of the universe of situations. Consequently, as an analysis of intuitionistic reasoning, it remains thoroughly classical. However, as an analysis of the reasoning of intelligent agents, it provides an initial framework for the global analysis of the agent's reasoning.

8.6.1.2 I-Structures and Validity

We will consider languages L with the logical symbols $\neg, \wedge, \vee, \rightarrow, \forall, \exists$, and $=$, together n -ary predicate symbols p, \dots , and function symbols f, \dots for various $n \geq 0$. The system LJA is as defined in Gentzen [2]. The system LJ' is defined as follows (cf. Prawitz [3]). Sequents $\Gamma \Rightarrow \Delta$ are permitted to have more than one formula in the succedent Δ . The axioms, structural rules, and the rules $\neg\neg IA$, $\wedge\neg IA$, $\vee\neg IA$, $\neg\neg IA$, $\forall\neg IA$, and $\exists\neg IA$, are just as for LK or LJ . The rules for introduction in the succedent are as shown in Figure 1.

$$\begin{array}{ll}
\rightarrow - IS : \frac{A, \Pi \Rightarrow B}{\Pi \Rightarrow \Sigma, A \rightarrow B} & \neg - IS : \frac{A, \Pi \Rightarrow}{\Pi \Rightarrow \Sigma, \neg A} \\
\wedge - IS : \frac{\Pi \Rightarrow A \quad \Pi \Rightarrow B}{\Pi \Rightarrow \Sigma, A \wedge B} & \vee - IS : \frac{\Pi \Rightarrow A}{\Pi \Rightarrow \Sigma, A \vee B} \quad \frac{\Pi \Rightarrow B}{\Pi \Rightarrow \Sigma, A \vee B} \\
\forall - IS : \frac{\Pi \Rightarrow A}{\Pi \Rightarrow \Sigma, \forall x A} & \exists - IS : \frac{\Pi \Rightarrow A_x[bfa]}{\Pi \Rightarrow \Sigma, \exists x A} \\
\text{provided the eigen-} & \text{where we use the} \\
\text{variable condition} & \text{substitution notation} \\
\text{is met} & \text{of Shoenfield[4].}
\end{array}$$

Figure 1: IS-Rules for LJ'

If $\Gamma'(\Delta')$ is a permutation of $\Gamma(\Delta)$, $\Gamma' \Rightarrow \Delta'$ is a *variant* of $\Gamma \Rightarrow \Delta$. Obviously any sequent provable in LJ is provable in LJ', and it is easy to prove the following lemma by induction on the complexity of proofs (cf. [3]).

Lemma 0.1 If $\Gamma \Rightarrow \Delta$ is provable in LJ', either $\Gamma \Rightarrow$ is provable in LJ or for some $A \in \Delta$, $\Gamma \Rightarrow A$ is provable in LJ.

Corollary 0.2 A sequent $\Gamma \Rightarrow A$ is provable in LJ if and only if it is provable in LJ'.

Def 0.3 If a_1, a_2, \dots , and b_1, b_2, \dots , are all terms of L, we will call formulas of the form $a = a$ *identity axioms* and we will call formulas of either of the two forms

$$\begin{aligned}
& a_1 = b_1 \wedge \dots \wedge a_n = b_n \rightarrow f(a_1, \dots, a_n) = f(b_1, \dots, b_n) \\
& a_1 = b_1 \wedge \dots \wedge a_n = b_n \rightarrow (p(a_1, \dots, a_n) \rightarrow p(b_1, \dots, b_n))
\end{aligned}$$

equality axioms, where in the latter, $p(a_1, \dots, a_n)$ could be $a_1 = a_2$.

Def 0.4 We will say that a sequent $\Gamma \Rightarrow \Delta$ is *provable* in LJ= if there exists a finite set Π of universal closures of equality and identity axioms such that $\Pi, \Gamma \Rightarrow \Delta$ is provable in LJ, and similarly for LJ'=.

Then we easily have:

Corollary 0.5 A sequent $\Rightarrow A$ is provable in LJ= if and only if it is provable in LJ'=.

Def 0.6 Now let R be a reflexive and transitive relation on the non-empty set K and let $k \in K$. A subset $C \subseteq K$ is a *world-line* from k if C is a maximal subset of K which is linearly ordered by R with first element k, and we will write $C \uparrow k$ to indicate this. Also, we write

$$R_k =_{dfn} R \upharpoonright \{k\} =_{dfn} \{k' \in K : k' R k\}$$

and

$$\check{R}_k =_{dfn} \check{R} \upharpoonright \{k\} =_{dfn} \{k' \in K : k R k'\}.$$

Def 0.7 A *semi-classical structure* \mathcal{A} for the language L consists of the following entities:

- a non-empty set $|\mathcal{A}|$, the *universe* of \mathcal{A} ;
- a binary function $\equiv : |\mathcal{A}|^2 \rightarrow \{U, V\}$;

- for each n -ary predicate symbol p , an n -ary total function $p_A : |A| \rightarrow U, V$;
- for each n -ary function symbol f , where $n > 0$, an n -ary partial function $f_A : |A|^n \rightarrow |A|$;
- for each individual constant c (i.e., 0-ary function symbol), an individual $c_A \in |A|$.

Moreover, we require that for any $a, b, c \in |A|$,

- $\equiv (a, a) = V$,
- if $\equiv (a, b) = V$, then $\text{equiv}(b, a) = V$, and
- if $\equiv (a, b) = V$ and $\equiv (b, c) = V$, then $\equiv (a, c) = V$.

We will often abbreviate $\equiv (a, b) = V$ by $a \equiv b$; thus $a \equiv b$ is an equivalence relation on $|A|$. We will generally write

$$A = \langle |A|, \equiv, p_A, \dots, f_A, \dots, c_A, \dots \rangle.$$

The values V and U can be thought of as signifying 'verified' and 'unverified', respectively. The language $L(A)$ is obtained from L by adding a new individual constant i_a to L for each $a \in |A|$; i_a is called the *canonical name* of a . Then A has a natural expansion to a semi-classical structure for $L(a)$, namely, set $(i_a)_A = a$ for each $a \in |A|$.

Def 0.8 Given the language L , an *I-structure* $A = \langle A_k, K, R \rangle$ for L consists of a reflexive and transitive relation R on a non-empty set K together with semi-classical structures for L ,

$A_k = \langle |A_k|, \equiv_k, p_k, \dots, f_k, \dots, c_k \rangle$, such that for all $k \in K$ (where we write p_k for p_{A_k} , etc.) each of the following hold:

1. if $k R k'$, then $|A_k| \subseteq |A_{k'}|$;
2. if $k R k'$ and $a, b \in |A_k|$, then $a \equiv_k b$ implies $a \equiv_{k'} b$;
3. if $k R k'$ and $a_1, \dots, a_n \in |A_k|$, then $p_k(a_1, \dots, a_n) = V$ implies $p_{k'}(a_1, \dots, a_n) = V$.
4. if $k R k'$, then $\text{dom}(f_k) \subseteq \text{dom}(f_{k'})$ and $f_{k'} \cap \text{dom}(f_k)^2 = f_k$;
5. if $k R k'$, then $c_k = c_{k'}$;
6. if $a_1 \equiv_k b_1, \dots, a_n \equiv_k b_n$, and if $\langle a_1, \dots, a_n \rangle$ and $\langle b_1, \dots, b_n \rangle$ are both in $\text{dom}(f_k)$, then $\bigwedge k' \in \check{R}_k \bigvee k'' \in \check{R}_k [f_{k''}(a_1, \dots, a_n) \equiv_{k''} f_{k''}(b_1, \dots, b_n)]$;
7. if $a_1 \equiv_k b_1, \dots, a_n \equiv_k b_n$, and if $p(a_1, \dots, a_n) = v$, then $\bigwedge k' \in \check{R}_k \bigvee k'' \in \check{R}_k [p_{k''}(b_1, \dots, b_n) = v]$.
8. $\bigwedge k \bigwedge a_1, \dots, a_n \in |A_k| \bigwedge k' \in \check{R}_k \bigvee k'' \in \check{R}_k [\langle a_1, \dots, a_n \rangle \in \text{dom}(f_{k''})]$.

We will always assume that

$$K \cap \bigcup_{k \in K} |A_k| = \emptyset.$$

For $S \subseteq K$, set

$$U(S) =_{dfn} \bigcup_{k \in S} |A_k|,$$

where $U(A) = U(K)$. Let Var be the set of free variables of L ; we will use x, y, z, \dots to range over Var .

Def 0.9 An *assignment* in A is a map $\nu : \text{Var} \rightarrow U(A)$.

Def 0.10 If ν is an assignment in A , $x \in Var$, and $a \in U(A)$, we define $\nu \left(\begin{smallmatrix} x \\ a \end{smallmatrix} \right)$ by

$$\nu \left(\begin{smallmatrix} x \\ a \end{smallmatrix} \right) (y) = \begin{cases} a & \text{if } x \text{ is } y \\ \nu(x) & \text{otherwise} \end{cases}$$

Also, set

$$\nu^\#(A) =_{dfn} \{x : x \text{ is free in } A\}.$$

Def 0.11 Let a be a term. The denotation of a in A at $k \in K$ relative to an assignment ν is given recursively:

$$x^{A,k}[\nu] \simeq \begin{cases} \nu(x) & \text{if } \nu(x) \in |A_k| \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$f_k(a_1, \dots, a_n)^{A,k}[\nu] \simeq \begin{cases} f_k(a_1^{A,k}[\nu], \dots, a_n^{A,k}[\nu]) & \text{if } a_i^{A,k}[\nu] \in |A_k| \text{ for } i = 1, \dots, n \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note that if $a^{A,k}[\nu]$ is defined, it lies in $|A_k|$.

Def 0.12 Given an assignment ν in A , $k \in K$, and a formula A , we define a satisfaction operator $A^{k,\nu}$ by recursion as follows (recall that V = 'verified' and U = 'unverified'):

1. $A^{k,\nu}(a = b) = \begin{cases} a^{A,k}[\nu] \equiv_k b^{A,k}[\nu] & \text{if } a^{A,k}[\nu], b^{A,k}[\nu] \text{ are both defined} \\ U & \text{otherwise} \end{cases}$
2. $A^{k,\nu}(pa_1 \dots a_n) = \begin{cases} p_k(a_1^{A,k}[\nu], \dots, a_n^{A,k}[\nu]) & \text{if } a_i^{A,k}[\nu] \text{ defined for } i = 1, \dots, n \\ U & \text{otherwise} \end{cases}$
3. $A^{k,\nu}(A \wedge B) = \begin{cases} V & \text{if } A^{k,\nu}(A) = A^{k,\nu}(B) = V \\ U & \text{otherwise} \end{cases}$
4. $A^{k,\nu}(A \vee B) = \begin{cases} V & \text{if } \bigwedge C \uparrow k \vee k' \in C [A^{k',\nu}(A) = V \text{ or } A^{k',\nu}(B) = V] \\ U & \text{otherwise} \end{cases}$
5. $A^{k,\nu}(\neg A) = \begin{cases} V & \text{if } A^{\ell,\nu}(A) = U \text{ for all } \ell \in \check{R}_k \\ U & \text{otherwise} \end{cases}$
6. $A^{k,\nu}(A \rightarrow B) = \begin{cases} V & \text{if } \text{cond}_1(A, B) \\ U & \text{otherwise} \end{cases}$

$$7. A^{k,\nu}(\forall xA) = \begin{cases} V & \text{if } \text{cond}_2(x, A) \\ U & \text{otherwise} \end{cases}$$

$$8. A^{k,\nu}(\exists xA) = \begin{cases} V & \text{if } \text{cond}_3(x, A) \\ U & \text{otherwise} \end{cases}$$

where we use the following abbreviations:

$\text{cond}_1(A, B)$ iff:

$$\bigwedge C \uparrow k \vee k' \in C \wedge \ell \in C [\nu^\# A \subseteq U(C) \& A^{\ell,\nu}(A) = V \Rightarrow \forall m \in C \cup \check{R}_\ell [A^{m,\nu}(B) = V]];$$

$\text{cond}_2(x, A)$ iff:

$$\bigwedge C \uparrow k \vee k' \in C \wedge \ell \in C \wedge a \in |A_\ell| \wedge m \in C \cap \check{R}_\ell [\nu^\# A \subseteq U(C) \Rightarrow A^{m,\nu} \left(\begin{smallmatrix} x \\ a \end{smallmatrix} \right) (A)]$$

$\text{cond}_3(x, A)$ iff:

$$\bigwedge C \uparrow k \vee k' \in C [A \subseteq C \Rightarrow \bigwedge \ell \in C \wedge a \in |A_\ell| [A^{\ell,\mu} \left(\begin{smallmatrix} x \\ a \end{smallmatrix} \right) (A)]].$$

Note that the mapping $A^{k,\nu}(A)$ is always defined. We say that A is *valid in A* if for each assignment ν in A and each world-line C in A (i.e., maximal subset of K linearly ordered by R) such that $\nu^\# A \subseteq U(C)$, there is a $k \in C$ such that $A^{k,\nu}(A) = V$. A sequent $A_1, \dots, A_n \Rightarrow B_1, \dots, B_m$ is *valid in A* if and only if the formula

$$A_1 \wedge \dots \wedge A_n \rightarrow B_1 \vee \dots \vee B_m$$

is valid in A . A formula or sequent is *valid* if and only if it is valid in all I-structures. The following lemmas are easy to verify by induction.

Lemma 0.13 Let A be a formula, let b and c be terms, let $k \in K$, and let ν be an assignment in A . Then:

$$1. (b_x[a])^{A,k}[\nu] \simeq b^{A,k}[\mu \left(\begin{smallmatrix} x \\ a^{A,k}[\nu] \end{smallmatrix} \right)];$$

$$2. A^{k,\nu}(A_x[a]) = A^{k,\nu} \left(\begin{smallmatrix} x \\ a^{A,k}[\nu] \end{smallmatrix} \right) (A).$$

Lemma 0.14 Let A be a formula, let a be a term, let $k, k' \in K$, and let ν be an assignment in A . Then:

$$1. \text{ if } kRk' \text{ and } a^{A,k}[\nu] \text{ is defined, then } a^{A,k'}[\nu] \text{ is defined and } a^{A,k}[\nu] = a^{A,k'}[\nu];$$

$$2. \text{ if } kRk' \text{ and } A^{k,\nu}(A) = V, \text{ then } A^{k',\nu}(A) = V.$$

8.6.1.3 Main Theorems

Theorem 0.15 (Validity Theorem) If $\Gamma \Rightarrow \Delta$ is provable in $LJ'_=$, then it is valid.

Theorem 0.16 (Completeness) A sequent $\Gamma \Rightarrow \Sigma$ is provable in $LJ'_=$ without cut iff it is valid in all I-structures in which every $|A_k|$ is finite.

References

- [1] Bowen, K.A., *A note on cut elimination and completeness in first order theories*, Zeit. F. math. Logik und Grund. d. Math., 18 (1972), 173-176.
- [2] Gentzen, G. *Investigations into logical deduction*, in *The Collected Papers of Gerhard Gentzen*, Amsterdam, 1969, 68-131.
- [3] Prawitz, D. *Some results for intuitionistic logic with second order quantification rules*, in *Intuitionism and Proof Theory*, Amsterdam, 1970, 259-269.
- [4] Shoenfield, J. *Mathematical Logic*, Reading, Mass., 1967.

8.6.2 Theoretical Semantics

The logic language on which metaProlog is based amalgamates an object-level language with its metalanguage. Consequently, attempts to adapt standard Tarskian semantics are extremely ugly, and semantic interpretations based on Kripkian semantics are only slightly better. A much more promising approach has arisen by taking two moves. First (related to the approach of A. Church's Theory of Types) is to regard *all* the expressions of the language as terms, with the formulas being merely a distinguished subclass of the terms. Second, one abandons the normal two-value truth value set and the normal "set of individuals" for the construction of denotations, and replaces them jointly with the set of all "reasonable" syntactic entities from the language itself, including partial proofs and search spaces. One then constructs a semantic interpretation using the so-called "substitutional interpretation", but attaches collections of partial proofs and partial search spaces to pairs of theories and formulas (the latter regarded as goals to be solved in the theory). It appears that many of the basic theorems of standard logic programming theory can be pushed through by brute force. However, it is much more appealing to attempt to adapt the ideas of Blair, Brown, and Subramanian which have shown that the basic theorems can be proved abstractly, given a suitable lattice structure on the space of truth values. The next step is the search for such a suitable lattice structure on the space of syntactic entities.

8.6.3 Reason Maintenance Experiment

We have been exploring several experimental knowledge-base management systems implemented using the metaProlog compiler(s). We exhibited a typical example at the RADC/NAIC Technology Fair during April, 1987. The top level of the system is sketched below. The primary predicates are

```
kbm(kb, int, mnt, kb_time)
```

```
react_to(request, kb, int, mnt, kb_time)
```


which are mutually tail-recursive. Three of the arguments are theories:

- kb – the domain knowledge base
- int – the theory defining integrity and consistency of kb
- mnt – the theory containing rules for revision and maintenance, together with data to effect revisions, etc.

The argument 'kb_time' simply is an abstract clock representation (here, system cycles – it could be real time). And the argument 'request' is simply the request for action obtained from the user: a query, a requested update, etc.

```
/*-----
    knowledge base manager main loop
    -- kbm and react_to are mutually tail-recursive
-----*/

all [kb, int, mnt, request, kb_time] :
    kbm(kb, int, mnt, kb_time)
    <-
        get_req(kb, request, kb_time) &
        react_to(request, kb, int, mnt, kb_time).

/*-----
    kbm primary action predicate: react_to
-----*/
/*=====
    Handling queries
    -- this simply returns one solution
       (prints the instantiated query)
    multiple solutions are handled by the "all [...]" request below
=====*/

all [kb, int, mnt, question, kb_time] :
    react_to(query(question), kb, int, mnt, kb_time)
    <-
        demo(kb, question) & ! &
        write('<<kbm: ') & write(question) & nl &
        kbm(kb, int, mnt, kb_time).

all [kb, int, mnt, question, kb_time] :
    react_to(query(question), kb, int, mnt, kb_time)
    <-
        ! &
        write('<<kbm--No solution: ') & write(question) & nl &
        kbm(kb, int, mnt, kb_time).

/*=====
    Queries requesting all solutions
    Input form is:
```

```

    all [x,y,z,...] : Formula
=====*/

all [kb, int, mnt, question, kb_time, vars, form, vars,
    realVars, instantiatedForm, sols, numVars] :
    react_to(all(vars, form), kb, int, mnt, kb_time)
    <-      ! &
        write('Trying all sols...') & nl &
        length(vars, numVars) &                               %create Prolog vars
        make_var_list(numVars, realVars) &
        subst_prolog(form, vars, realVars, instantiatedForm) &
        %instantiate Formula
        (demo(kb, setof(realVars, instantiatedForm, sols)) & ! &
        write('Solutions found:') & nl &
        show_list(sols);
        %% No solutions alternative
        write('No solutions found...') & nl)
        & kbm(kb, int, mnt, kb_time).

/*=====
    React to requests to add an assertion to the kb
=====*/

all [kb, int, mnt, assertion, vars, var_list, form, inst_form, new_mnt,
    new_kb, kb_time, new_kb_time] :
    react_to(add(assertion), kb, int, mnt, kb_time)
    <-      ! &
        (assertion = ':'(all(vars), form) & ! &
        instantiate(assertion, inst_form, var_list) &
        check_conseqs(inst_form, var_list, kb, int, mnt,
            new_kb, new_mnt, assertion, kb_time);
        integ_ck(assertion, kb, int, mnt, new_kb, new_mnt, kb_time)) &
        new_kb_time is kb_time + 1 &
        kbm(new_kb, int, new_mnt, new_kb_time).

```

The subsidiary predicates of interest above are those dealing with the checking of consequences, integrity, and consistency, defined as follows.

```

all [form, var_list, kb, int, mnt, new_kb, new_mnt, assertion,
    inst_form, kb_time] :
    check_conseqs(inst_form, var_list, kb, int, mnt,
        new_kb, new_mnt, assertion, kb_time)
    <-
        (work_thru_conseqs(inst_form, var_list, kb, int, mnt,
            new_kb, new_mnt, assertion, kb_time) & !;
        write('Denying addition of assertion to the kb...') & nl &
        new_kb = kb &
        addto(mnt, failed_add(assertion,kb,int,mnt,kb_time), new_mnt)).

all [inst_form, kb, int, mnt, new_kb, new_mnt, assertion, kb_time,

```

```

    head, body, exist_quant_body, var_list, head_list, head_vars,
    body_vars] :
work_thru_conseqs(inst_form, var_list, kb, int, mnt,
    new_kb, new_mnt, assertion, kb_time)
<-
    (inst_form = (head :- body) & ! &
        vars_occurring_in(head, head_vars) &
        difference(var_list, head_vars, body_vars) &
        exist_quant(body_vars, body, exist_quant_body) &
        (demo(kb, setof(head, exist_quant_body, head_list)) &
            show_list(head_list,3);
        write('No immediate head consequences of this assertion...') &nl);
        write('The expression ') & write(inst_form) &
        write(' is not an implication...ignoring for now...') &nl) &
    addto(mnt, added(assertion, kb, int, mnt, kb_time), new_mnt) &
    addto(kb, inst_form, new_kb).

all [kb, int, mnt, assertion, new_kb, new_mnt, kb_time] :
    integ_ck(assertion, kb, int, mnt, new_kb, new_mnt, kb_time)
<-
    (demo(kb, assertion) & ! & write('Duplication--nothing added') & nl ;
        demo(int+kb, acceptable(assertion)) &
        contradict_check(assertion, kb, int) &
        write('Integrity check passed...') & nl &
        addto(mnt, added(assertion, kb, int, mnt, kb_time), new_mnt) &
        addto(kb, assertion, new_kb)).

all [kb, int, mnt, assertion, new_kb, new_mnt, kb_time, body,
    ans, ans1, inst_body,d,e,f,g] :
    integ_ck(assertion, kb, int, mnt, new_kb, new_mnt, kb_time)
<-
    not(demo(int,clause(acceptable(assertion), body))) &
    (demo(int+kb, update_via(assertion, body)) &
        write('Assertion to add is defined by the following view:')
        & nl & nl &
        write(' ') & write(assertion) &
        write(' if ') & write(body) & put('.') & nl & nl &
        write('Do you want to attempt the addition via this view?') &
        read(ans1) &
        (affirmative(ans1) & ! &
            (not(body = (d,e)) &
                demo(int+kb, acceptable(body)) & !; true) & ! &
                (not(body = (f,g)) &
                    contradict_check(assertion, kb, int) & !; true) &
                    write('Integrity check passed...') & nl &
                    check_instances(body, inst_body),
                    addto(mnt,added(inst_body, kb,int,mnt,kb_time), new_mnt) &
                    addto(kb, inst_body, new_kb); fail);
                %otherwise for demo(int+kb, update....)
            contradict_check(assertion, kb, int) &
            write('No (other) integrity clauses apply to ') &

```

```

        write(assertion) & nl &
        write('Do you want to accept it as a pure premise?') & read(ans) &
        integ_act_on(ans, assertion, kb, int, mnt, new_kb, new_mnt, kb_time)).

all [kb, int, mnt, assertion, new_kb, new_mnt, kb_time, ans] :
    integ_act_on(ans, assertion, kb, int, mnt, new_kb, new_mnt, kb_time)
    <-
        affirmative(ans) &
        write('Adding premise: ') & write(assertion) & nl &
        addto(mnt, added(assertion, kb, int, mnt, kb_time), new_mnt) &
        addto(kb, assertion, new_kb).

all [kb, int, mnt, assertion, new_kb, new_mnt, kb_time, ans] :
    integ_act_on(ans, assertion, kb, int, mnt, kb, new_mnt, kb_time)
    <-
        not(affirmative(ans)) &
        write('Denying addition of premise: ') & write(assertion) & nl &
        addto(mnt, failed_add(assertion, kb, int, mnt, kb_time), new_mnt).

all [kb, int, mnt, assertion, new_mnt, kb_time] :
    integ_ck(assertion, kb, int, mnt, kb, new_mnt, kb_time)
    <-
        write('Integrity check failed for ') & write(assertion) &
        write(' at time ') & write(kb_time) & nl &
        addto(mnt, failed_add(assertion, kb, int, mnt, kb_time), new_mnt).

all [assertion, kb, int, contrary] :
    contradict_check(assertion, kb, int)
    <-
        demo(int, contradictory(assertion, contrary)) &
        & demo(kb, contrary)
        & ! & fail.

all [assertion, kb, int] :
    contradict_check(assertion, kb, int).

```

The general knowledge-base management machinery built up above was applied to a small application concerning the NAIC consortium. First we need a starting knowledge base. This could have begun empty. The overall system provides facilities for saving the current state of the kbm system.

```

theory(naic_kb).      % the knowledge

located(su, city(syracuse)).
located(city(syracuse), state(ny)).
located(ub, city(buffalo)).
located(city(buffalo), state(ny)).
located(um, city(amherst)).
located(city(amherst), state(mass)).

pi(person(lesser, vic), project(1)).

```

```

pi(person(croft,bruce), project(1)).
title(project(1),[a,knowledge,acquisition,assistance,and,explanation,system]).

pi(person(bowen,ken), project(2)).
title(project(2),[knowledge,base,maintenance]).

pi(person(shapiro,stu), project(3)).
title(project(3), [a,versatile,expert,system,for,equipment,maintenance]).

located(radc, afb(griffiss)).
located(afb(griffiss), state(ny)).

all [x,y] : in(x,y) <- located(x,y).
all [x,y,z] : in(x,y,z) <- located(x,z), in(z,y).

all [x,y,z] : same_state(x,y) <- in(x,state(z)) & in(y, state(z)).

all [person, title, project] :
    directs(person, title)
    <-
        pi(person, project) & title(project, title).

endtheory.

```

Next we need definitions of predicates which provide for integrity and consistency maintenance. There are several points worth noting. First, the definition of 'acceptable' has only one argument, namely the proposed new addition to the knowledge. However, examination of the code for 'integ_ck' shows that the goal

```
acceptable(Update)
```

is run in the context of the current state of the knowledge base (combined with the theory 'naic_int' defined below). Thus, this effectively defines the notion of 'acceptable with respect to the current knowledge base'. Secondly, the definitions of 'acceptable' and 'inconsist' are domain-specific: They apply to the anticipated assertions which the system may consider. Finally, note that the knowledge base designer can define procedures (possibly domain-specific) which update derived views under acceptable circumstances, as seen in 'rec.update_via'.

```
theory(naic_int).    meta-level integrity & consistency
```

```

all [place1, place2] :
    acceptable(located(place1, place2))
    <-
        subsidiary(place1, place2).

all [place1, place2] :
    subsid(place1, place2) <- atom(place1).

```

```

all [place1, place2] :
    subsid(city(place1), state(place2)).

all [place1, place2] :
    subsid(state(place1), country(place2)).

all [place1, place2] :
    subsid(country(place1), continent(place2)).

all [place1, place2] :
    subsidiary(place1, place2)
    <-
        subsid(place1, place2).

all [place1, place2, place3] :
    subsidiary(place1, place2)
    <-
        subsid(place1, place3) & subsidiary(place3, place2).

all [x] :
    is_place(city(x)) <- atom(x).

all [x] :
    is_place(state(x)) <- atom(x).

all [x] :
    is_place(country(x)) <- atom(x).

all [x] :
    is_place(continent(x)) <- atom(x).

all [name1, name2, number, boss, what] :
    acceptable(pi(boss, what))
    <-
        boss = person(name2, name1)
        & what = project(number) & integer(number).

all [number, words] :
    acceptable(title(project(number), words))
    <-
        integer(number) & list_of_atoms(words).

list_of_atoms([]).

all [head, tail] :
    list_of_atoms([head | tail])
    <-
        atom(head) & list_of_atoms(tail).

%%===== Rules for inconsistency ===== %%%

all [x, y] :

```

```

contradictory(x, y)
<-
  inconsist(x, y).

all [x, y] :
  contradictory(x, y)
  <-
    inconsist(y, x).

all [x] :
  inconsist(male(x), female(x)).

all [x] :
  acceptable(male(x)).

all [x] :
  acceptable(female(x)).

%%===== Rules for updating views ===== %%%%

all [assertion, definition, body] :
  update_via(assertion, definition)
  <-
    clause(assertion, body) &
    rec_update_via(body, definition).

all [a1,a2, b1,b2] :
  rec_update_via((a1,a2), (b1, b2))
  <- ! &
    rec_update_via(a1, b1) & rec_update_via(a2, b2).

all [assertion, definition] :
  rec_update_via(assertion, definition)
  <-
    var(assertion) & ! & fail.

all [assertion, definition] :
  rec_update_via(assertion, definition)
  <-
    atom(assertion) & ! &
    (clause(assertion, body) & ! &
    rec_update_via(body, definition) ;
    definition = assertion).

all [assertion, predicate, args] :
  rec_update_via(assertion, assertion)
  <-
    assertion =.. [predicate | args].

endtheory.

theory(naic_mnt).    % kbm maintenance

```

```

all [x,y] :
    subset(x, y)
    <-
        subset0(x,y).

all [x,y,z] :
    subset(x, y)
    <-
        subset0(x,z) & subset(z, y).

all [x, y, z] :
    belongs(x,y)
    <-
        belongs_to(x, y, z).

all [x, y, z, u] :
    belongs(x, y)
    <-
        true &
        subset0(u, y) &
        demo(y, clause(x, z)).

endtheory.

```

While only a small toy example, the code above demonstrates the ease with which knowledge base implementers can directly define notions of consistency and maintenance specific to the content of the particular application. (No revision maintenance is conducted in this example. See the 'ltm' example below.)

8.6.4 Reason Maintenance and Theory Manipulation

By bringing the problem-solver and knowledge-base maintenance program closer together than previously, a very useful and potentially efficient methodology has evolved. As in the preceding section, one writes a version of the metaProlog interpreter as normally expressed in Prolog with an explicit argument indicating the theory under which the deduction is being performed. However, in this interpreter, one includes an *implementation* of the usual Prolog assert. At the metalevel, this is a logical axiomatization of such a system. The implementation of assert requires that the interpreter check the consistency of the assertion being added against an integrity theory which is also carried around by the interpreter. If the consistency check fails, the interpreter consults its revision theory to guide revision of the knowledge base to a consistent state. As is usual with such interpreters, a source-to-source transformer is created which partially evaluates the domain problem-solving rules and knowledge base relative to this interpreter (via expansion of arguments). Not only do the transformed rules run much more efficiently, but consideration of the manner in which they are compiled provides insight into how the process might be pushed deeper into the metaProlog compiler's abstract machine.

Some exploratory work on an implementation (in metaProlog) of a "logic-based" reason maintenance system in the style of McAllester was begun. A top-level sketch of this experiment follows. (It is written in conventional Prolog syntax which our metaProlog compiler accepts. When more fully developed, it will be converted to metaProlog syntax via an automatic conversion program.)


```

/*-----
|           ltm.pro
|           Logic-Based Truth Maintenance
*-----*/

solve(Problem, Solution, KB, Final_KB)
:-
    demo(solver, solved(KB, Solution^Problem) ).

x(N) :-
    name(N, N_String),
    append("examp", N_String, ".pro", File_String),
    name(File, File_String),
    append("x", N_String, Th_Name_String),
    name(Theory_Name, Th_Name_String),
    consult(File, Theory_Name),
    demo(Theory_Name, initialize_kb(KB) ),
    demo(Theory_Name, goal_problem(Problem/Solution) ),
    solve(Problem, Solution, KB, Final_KB),
    nl, write(problem=Problem),nl,
    write(' solved by solution:'),nl,
    write(Solution),nl.

theory solver. % $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

use(kb_primitives).

solved(KB, Output^Problem, T) :-
    current_focus(KB, T0),
    solves(Problem, KB, T0, T).

solves(Problem, KB, KB, T, T)
:-
    status(Problem, KB, T, true), !.

solve( (Prob1 & Prob2), KB0, KB1, T0, T1)
:- !,
    solves(Prob1, KB0, KB_Inter, T0, T_Inter),
    solves(Prob2, KB_Inter, KB1, T_Inter, T1).

solves(Problem, KB0, KB1, T0, T1)
:-
    rule_of(KB0, T0, Problem, Body),
    solves(Body, KB0, KB1, T0, T1).

solves(Problem, KB0, KB1, T0, T1)
:-
    status(Problem, KB0, T0, unknown),
    possible_assumption(KB0, T0, Problem),
    demo(reason_maint, assumable(Problem, KB0, KB1, T0, T1) ).

```

```

/*****
Deep failure causing backtracking to this point will be handled
tail-recursively inside 'reason_maint' in the definition of
assumable, which will look for an acceptable way to back up
the theory T0 (typically removing some assumptions under some
maintenance regime) to yield a theory T3 and knowledge base
state KB3, and then calling
solves(Problem, KB3, KB1, T3, T1).
Consequently, we see that if we originally submit the goal
:-solves(Problem, KB0, KB1, T0, T1)
and it succeeds, T1 is not necessarily a monotonic extension of T0,
but is an extension of some acceptable revision of T0.
*****/

endtheory. % solver #####

theory reason_maint. % #####

use(kb_primitives).

assumable(Formula, KB0, KB1, T0, T1)
:-
    status( not(Formula), KB0, T0, true), !,
    find_alterate(Formula, KB0, T0, KB3, T3),
    demo(solver, solves(Formula, KB3, KB1, T3, T1) ).

assumable(Formula, KB0, KB1, T0, T1)
:-
    integrity_theory_of(KB0, Integ_Th),
    demo(Integ_Th, acceptable(Form, T0) ),
    not( inconsistent(Formula, T0, KB0) ),
    addto(T0, Formula, T1),
    update_maint_records(T0, addto(T0, Formula, T1), T1, KB0, KB1).

assumable(Formula, KB0, KB1, T0, T1)
:-
    find_alterate(Formula, KB0, T0, KB3, T3),
    solves(Formula, KB3, KB1, T3, T1).

inconsistent(Formula, Theory, KB)
:-
    status(Formula, Other_Theory, KB, false),
    records(KB, extends(Other_Theory, Theory) ), !.

inconsistent(Formula, Theory, KB)
:-
    demo(Theory, not(Formula) ).

endtheory. % reason_maint #####

theory kb_primitives. % #####

```

```

records(KB, extends(T1, T2) )
:-
    extension_records_of(KB, Ext_Recs),
    demo(Ext_Recs, extends(T1, T2) ).

update_maint_records(T0, addto(T0, Formula, T1), T1, KB0, KB1)
:-
    extension_records_of(KB0, Ext_Recs0),
    addto(Ext_Recs0, extends(T0, T1), Ext_Recs1),
    update_kb(extension_records, Ext_Recs1, KB0, KB1).

status(Formula, Theory, KB, Status_Value)
:-
    kb_access(status_records, KB, Status_Theory),
    demo(Status_Theory, status(Theory, Status_Value) ).

extension_records_of(KB, Ext_Recs)
:-
    kb_access(extension_records, KB, Ext_Recs).

current_focus(KB, T)
:-
    kb_access(current_focus, KB, T).

kb_access(What, KB, Ext_Recs)
:-
    kb_access_table(What, ArgNum),
    arg(ArgNum, KB, Ext_Recs).

kb_vector_size(3).

kb_access_table(status_records, 1).
kb_access_table(extension_records, 2).
kb_access_table(current_focus, 3).

make_kb(Arg_List, KB)
:-
    kb_vector_size(KB_Size),
    functor(KB, kb, KB_Size),
    install_kb_args(Arg_List, KB).

install_kb_args([], KB).

install_kb_args([ Entry_Name = Entry_Value | Rest_Arg_List], KB)
:-
    kb_access_table(Entry_Name, Entry_Num),
    arg(Entry_Num, KB, Entry_Value),
    install_kb_args(Rest_Arg_List, KB).

endtheory. % kb_primitives #####

```

In the course of working on the ltm example, it is becoming clear that our approach to metaProlog may provide an even greater potential with regard to reason maintenance than we originally thought. When one reflects on the details of our metaProlog compiler, one sees that there are two distinct components to the treatment of theories: (1) provision for the raw physical notion of individual independent theories; in our system, theories are identified with clause-indexing patches referring to subsets of a global "blackboard" of clauses; (2) maintenance of relationships among theories; in the present metaProlog, we support maintenance of historical relationships.

The implementation of the former is independent of the latter (though 2 does rely on 1, but this causes no problem for the following). It seems apparent that (2) could be replaced or supplemented by maintenance of other sorts of relationships between theories, in particular, the sorts of relationships inherent in reason maintenance.

A good deal of more exploration and experimentation will be necessary before the situation becomes sufficiently clear to determine whether the processes of reason maintenance can be sufficiently analyzed into primitive process to warrant elaboration of additional instructions and facilities in the underlying Abstract Prolog Machine.

Assumption-Based Reason Maintenance was also a concern. Here the primary concern is with the excessive storage demands of de Kleer's methods. The goal is to discover methods of achieving much more virtual implementations of his ideas. The target is to be able to explicitly maintain the theories lying along the two fringes: The boundary between the unexamined theories and the known inconsistent theories, and the boundary between the unexamined theories and the known consistent theories. All other theories which have been examined (both consistent and inconsistent) will be maintained in as compact a virtual representation as possible by describing them in terms of theories lying on the fringe. In essence, these descriptions say what must be added to or deleted from a fringe theory in order to obtain a given virtual theory. The advantage can be gained by grouping the theories around the statements. Conceptually, the maintenance will involve quadruples

$v(\text{Formula}, \text{Sign}, \text{FringeTheory}, L),$

where L is a list of virtual theory ids such that the given formula must be added to (Sign = +) or deleted from (Sign = -) FringeTheory for each of the theories whose id is on the list.

8.7 Ilyas Cicekli:

metaProlog Implementation

The MetaProlog system described in this report is a compiler-based meta-level system for the MetaProlog programming language. The Warren Abstract Machine (WAM) is extended to the Abstract MetaProlog Engine (AMPE) for MetaProlog. Moreover, a MetaProlog program is directly compiled into the instructions of the AMPE.

8.7.1. Introduction

Most of the meta-level systems implemented in last decade are meta-level interpreters which introduce extra interpretation layers which slow down the execution. The MetaProlog system described in this report is a compiler-based meta-level system for the MetaProlog programming language. Since MetaProlog is an extension of Prolog, we extended the Warren Abstract Machine (WAM) to the Abstract MetaProlog Engine (AMPE). MetaProlog programs are directly compiled into the instructions of the AMPE.

In the rest of this report, the MetaProlog system is briefly described. Theories which are the first class objects in MetaProlog, and their representations in the MetaProlog system is discussed in Section 2. Then the basic structure of the AMPE is explained in Section 3. In the last two sections, the garbage collector and the debugger of the MetaProlog system are presented respectively.

8.7.2. MetaProlog Theories

In Prolog, there is a single database, and all goals are proved with respect to this database. When there is a need to update this database, the builtins `assert/retract`, which are ad hoc extensions to the basic logic programming paradigm, are used to create the new version of this database by destroying the old database in the favor of the new one. On the other hand, there can be more than one theory in MetaProlog, and a goal can be proved with respect to one of these theories. A new theory in MetaProlog is created from an old theory without the destroying the old theory.

A new theory is created from an old theory in the system by adding some clauses or dropping them. The new theory inherits all procedures of the old theory except for procedures explicitly modified during its creation. Although we create a new theory from an old theory, the old theory can still be accessed by the system.

The provability relation between a theory and a goal is explicitly represented in MetaProlog by a two argument predicate `demo`. The relation `demo(Theory,Goal)` holds precisely when `Goal` is provable in `Theory`. Similarly, the relation `demo(Theory,Goal,Proof)` holds when `Proof` is the proof of `Goal` in `Theory`. When one of these provability relations is encountered, the underlying theorem prover tries to prove the given goal with respect to the given theory.

Theories of the MetaProlog system are organized in a tree whose root is a distinguished theory, the base theory. The base theory contains all the system builtins, and all other theories in the system are descendants of the base theory. In other words, all theories can access procedures of the base theory.

Every theory in the MetaProlog system possesses a default theory except for the base theory. The default theory of a theory `T` is the theory where we search for a procedure if the search for that procedure in `T` fails. This search through default theories continues until the procedure is found or the base theory is reached.

To shorten the depth of the theory, theories in the MetaProlog system are classified into two groups : default theories, and non-default theories. A non-default theory is a theory that carries information about all procedures that underwent modifications in the ancestor theories between this theory and its default theory. Access to these procedures is very fast, at expense of copying some references. The default theory of a theory is the first ancestor theory that is a default theory. A default theory is a theory whose descendants don't carry any information about the procedures

occurring in that theory. If only default theories are used, access to a given procedure in a given theory may require a search through all its ancestor theories. In this case, access to a procedure may be slow, but no copying of references is needed. Depending on the problem, the system tries to use one or the other approach, or a combination of both to achieve a balance between speed of access and space overhead.

When a new theory is created from a non-default theory, its default theory will be its father's default theory. But if a new theory is created from a default theory, its default theory will be its father. In the first case, the new theory will be at its father's level. In the second case, the new theory will be at one level above its father's level. Thus we don't increment the depth of the theory tree when a theory is created from a non-default theory.

8.7.3. Abstract MetaProlog Engine

Our main goal in this project was to create an efficient compiler-based MetaProlog system. Since MetaProlog is an extension of Prolog, the Warren Abstract Machine (WAM) was the best starting point. For this purpose, the WAM is extended to the Abstract MetaProlog Engine (AMPE).

The AMPE performs most of the functions of the WAM, but it also has some extra features to handle theories and compiled procedures as data objects of the system. These extra features basically cover extra registers to handle theories and a different memory organization.

The code space and the heap in the WAM is integrated as a single data area in the AMPE which is more suitable to handle compiled procedures as data objects. This integrated space in the AMPE is still called the heap. Thus theories and compiled procedures can be created on fly, and they are can be easily discarded when the need for them is gone. The local stack and the trail of the AMPE still perform the same job they perform in the WAM.

There are two new registers in the AMPE in addition to the registers the WAM uses. The first one is the theory register which holds the current theory (context) of the MetaProlog system. The value of the theory register is changed when the context of the system is switched to the another context. This register is also saved in choice points so that the context of the system can be restored the value saved in the last choice point during backtracking. The second one is the theory counter register which is simply a counter to produce a unique theory-id for each theory in the system. It is incremented to indicate the next available theory-id after each creation of a new theory.

The AMPE can run in two different modes. When a two argument "demo" predicate is encountered, the system runs in the simple mode. In the simple mode, the system only proves a goal with respect to the current theory of the system. When a three argument "demo" predicate is encountered, the mode of the system is switched to the proof mode. In the proof mode, a goal is not only proved with respect to the current theory of the system, its proof is also collected. At the implementation level, the mode of the system is represented by a mode flag which is also saved in choice points so that the system can switch from one mode to the another during backtracking.

In the simple mode of the system, only the core part of the system described above is used. On the other hand, two extra registers are used in addition to the core part of the system when the system runs in the proof mode. These extra two registers are used to collect the proof of a goal during its execution.

8.7.4. Garbage Collector

The garbage collector of the MetaProlog system collects all the garbage in the system including the garbage in the code. It consists of a recursive marking routine and a compaction routine. The marking routine recursively marks all locations in the heap which are accessible from external locations such as argument registers, and locations in the local stack. The garbage compaction routine, an extension of Morris's compaction algorithm, adjusts all pointers in the uncompact heap and does the real compaction.

8.7.5. Debugger

The debugger for the MetaProlog system is similar to a four-port debugger for a Prolog system. The debugger itself is written in MetaProlog. The debugger always lives in the base theory so that it can be reachable from every theory in the system at any time.

8.8 Keith Hughes¹

Interfaces to Databases

8.8.1. Introduction

The combination of logic programming and relational database systems is a desirable goal, because intelligent processing of large numbers of facts becomes possible. Database systems are very good at retrieving large amounts of data while doing little or no inference. On the other hand, logic programming languages such as Prolog provide powerful methods for doing inference, but are inadequate when it comes to processing substantial bodies of facts.

Logic and relational database systems (RDBS) are known to have close theoretical connections [Gallier78]; and many people have advocated an amalgam of the two. Extensions to Prolog to achieve such an amalgamation have been suggested, but there are problems with each. They do, however, point to possible solutions. VMProlog allows SQL queries to be used in the middle of Prolog statements, but this makes a distinction between program and data. The resulting programs are overly complicated. Other methods, which require direct modifications to Prolog itself, include the **compiled method** [Reiter78a] and the **interpretive method** [Minker78].

The system to be described here is a combination of a Prolog system and the RDBS system Ingres [Stonebraker76]. This system provides a framework for experimentation with alternatives for handling the interface between Prolog and a RDBS. It uses a variant of the compiled approach to hand queries to the RDBS system. The database system is extended with a secondary program to handle the large amounts of data. This paper will trace the history of the system, with particular attention to the problems which arose, and what was done to solve them. Finally, a plan for future work will be given.

8.8.2. Previous Approaches

The previous approaches have encountered a variety of problems. These problems include such areas as handling recursion, efficient handling of very large databases (of the order of gigabytes of information), and readability of programs. This section discusses some of the major attempts and the problems related to each.

8.8.2.1. The Interpreted Method

The interpreted method [Minker78] requires major changes to the underlying Prolog system. The reader is referred to [Chakravarthy] for the details of this method. The major idea is that the computation extracts the correct answer from the set of all possible solutions to each subgoal of the program. Each subgoal is seen as a restriction process. All of the possible answers from the previous state of the query are examined by the current restriction, and those not passing are removed. This method, instead of being the one-answer-at-a-time idea that Prolog adopts, provides the user with all of the answers at once.

The main problem is the amount of data that must be passed between the logic system and the RDBS, especially if they are in different processes on a single machine, or split up between two pieces of hardware. The database is going to send megabytes of information to Prolog, which will then pass it back. Much time is going to be spent in communication of this data. Moreover, buffering this much data in the two systems requires Prolog to have a database manager of its own, defeating the purpose of using the database system in the first place. Techniques exist for optimizing the data structure representing the set of solutions at each stage of the computation, but this method is felt to be inadequate for very large databases.

8.8.2.2. The Compiled Method

¹ This work supported by Applied Logic Systems, Inc. under U.S. Army contract nnn.

The compiled method [Reiter78a] postpones database queries as long as possible before sending them to the database system. A meta-interpreter could be written with definite clause grammars in Prolog to simulate this method. The interpreter would notice when a database call is being made and add it to a list of other calls that are pending. When the main program finishes running, all of these queries are sent in bulk to the database system, at which time the user gets the answer to his query back.

The problem with this method is that it is assumed that the procedures in the logic program are non-recursive. When the program is recursing, the system could possibly pile up requests until memory was full, getting no useful work done. [Reiter78b] discusses cases where the recursion terminates, and takes advantage of this. However, not every program will have this ability. Some recursions may terminate only when an appropriate answer is retrieved from the database. One possibility is to perform a flow analysis on the program and decide when recursion will not terminate, and make the database calls earlier. However, the flow analysis may prove to be difficult.

8.8.2.3. VMProlog

VMProlog is closest to the current method used. VMProlog allows statements to be made to the SQL database system by interspersing SQL statements with Prolog goals. An evaluable predicate SQL was added to Prolog, which allows a query to be sent to the appropriate system. An example call to SQL/DS would be

```
...,sql('select flynb,airport2 from flyex where airport1="ROME"',*1),...
```

where *1 is the variable to be instantiated to a list of the answers to the query. The query is allowed to backtrack if necessary, giving more possible instantiations of the variable.

This interface allows the database call to look like a Prolog call, but the statement of the query is not the same as if the query were stated as

```
$...,flyex(Flynb,rome,Airport2,\_,\_,\_),...$
```

which looks more like a Prolog predicate. A simple database compiler could take care of this problem.

8.8.3. The Syracuse Implementation

The system which follows is a testbed for trying various alternatives for Prolog/RDBS interfaces. Several evaluable predicates were added to a version of Prolog written at Syracuse [Bowen85] to allow communication with the Ingres RDBS. Having no large databases to test the system has been a problem. The Prolog system and the interface were written in C on a VAX780 running Unix.

A variation of the compiled approach and the VMProlog approach is used. The SQL predicate in VMProlog allows the user to keep writing in Prolog without having to delve too far into another language. The compiled method is advantageous in that it requires only slight modification to existing Prolog systems and doesn't have the problems the interpreted method has with very large databases.

8.8.3.1. The Initial Attempt

The first pass at the interface added three new predicates to Prolog: *initIngres*, *callIngres*, and *killIngres*. *initIngres* started up an Ingres sub-process, which could then be removed by *killIngres*. *initIngres* had one argument, being the database in which the predicates were to be found.

callIngres actually made the queries to Ingres through the EQUQL [Stonebraker76] routines supplied with Ingres. EQUQL supplies a series of C routines to allow an application programmer to call Ingres from the application program. *callIngres* had a single argument, being the predicate the

user was interested in. This call was then changed into the QUEL query language for Ingres and sent to Ingres, where it was processed. The results were then returned and unified with the variables in the call. Any atoms retrieved by Ingres were installed in Prolog's name table.

For example, a call of

```
callIngres(parts(PNum,PName,pink,Weight,Qoh))
```

would cause the QUEL statements

```
range of e is parts
retrieve (e.pnum,e.pname,e.weight,e.qoh) where e.color="pink"
```

to be sent to Ingres. If no tuples were returned, *callIngres* would fail. If there were any answers, *callIngres* would unify the variables in the call with the answers returned. If backtracking occurred, the next tuple would be retrieved, and the variables in the call would be re-bound.

This approach has several problems. First, Prolog has to know about the details of QUEL. If another database system were to be used, the evaluable predicates would have to be rewritten. Second, Ingres returns all solution tuples at once, while Prolog can only consume one answer at a time. Finally, there is only one communication channel out of Ingres. Thus, the user is allowed only one backtrackable call to the RDBS.

8.8.3.2. DBMachine

DBMachine is an program to handle two of the problems encountered with *callIngres*: (1) the RDBS wants to retrieve all tuples answering a query at once, and (2) the need for more than one call to the database at a time, with backtracking if necessary. It is a program which allocates buffers to Prolog calls to the database, passes the call to Ingres, and stores the tuples received from Ingres in these buffers.

When Prolog needs the database system, it creates a DBMachine process, which then starts up an Ingres process with the appropriate database. Prolog requests are made to DBMachine, which gets the required information from Ingres through calls to EQUEL. Splitting up the processes is useful in seeing how to handle networks of machines talking to each other.

No QUEL statements are sent by Prolog. A much simpler request language is used by Prolog, which then can be translated to any RDBS query language, such as SQL or QUEL. This was done to increase the communication bandwidth between Prolog and the RDBS. Also, the Ingres dependence was taken away from Prolog, allowing DBMachine to call any database system without modifications to the evaluable predicates in Prolog.

8.8.3.2.1. The Prolog/DBMachine Interface

Development of this interface went through two phases. The overall appearance didn't change much, but the underlying mechanisms changed.

The applications programmer talks to DBMachine through three predicates: *initDB*, *killDB*, and *queryDB*. The first two predicates are analogous to *initIngres* and *killIngres*, except that they start and stop DBMachine.

queryDB(Query) passes a form of *Query* to DBMachine and instantiates any variables found in *Query* to the tuples passed back from DBMachine. *queryDB* itself is not the actual call to DBMachine, however. In order to keep the routines in C from becoming unmanageable, and to allow Prolog to backtrack over the stream of tuples that DBMachine has generated in response to the query, *queryDB* is written in Prolog as follows:

```

queryDB(Query) :-
    requestDB(Query, BufferID),
    getAnswers(g(BufferID), Query).

getAnswers(ID, Query) :-
    answerDB(ID, Query).
getAnswers(g(BufferID), _) :-
    BufferID < 0,
    !, fail.
getAnswers(ID, Query) :-
    getAnswers(ID, Query).

```

Notice that the two predicates called *requestDB* and *answerDB* are hidden from the casual user because they have a procedural flavor, whereas the top level call *queryDB* does not. *requestDB* is called with two arguments, one instantiated to the query to be made, and the second, a variable to be instantiated to the number (a positive integer) of the buffer in DBMachine where the results will reside. The results are then retrieved from the buffer by *answerDB*, which is encapsulated in *getAnswers* to allow Prolog to backtracking over the query. The internals of *queryDB* are hidden from the user so that *answerDB* can make a destructive assignment to signal when no more answers are to be found in the buffer. If there are tuples left in the buffer, *answerDB* will set the variables in *Query* to their proper values for the next tuple in the buffer. If the program requires more answers for *Query*, the first and second clauses will fail, and *getAnswers* will be called again. The combination of the first and third clauses for *getAnswers* effects the iteration through the buffer as Prolog backtracks over *Query*. If the buffer is empty, *answerDB* changes the *BufferID* to -1 and fails. This is caught by the second clause of *getAnswers*, which causes *getAnswers* to fail. Any atoms retrieved through *answerDB* are placed in Prolog's name table.

requestDB modifies *Query* in order to limit the amount of information sent to DBMachine. In the first phase of the interface development, this information consisted of the predicate name, a list of numbers describing which columns contained the uninstantiated variables, and a list of column numbers corresponding to instantiated values, along with those values. *answerDB* would examine the Prolog structure of *Query*, noticing where the variables were, and get the corresponding tuple values back from the buffer in DBMachine.

8.8.3.2.1.1. Problems with Phase One and Their Solution

Phase one of this interface only allowed the user to hand one call to DBMachine at a time. To solve goals of the form

```
..., a(A, B, C), b(F, A, R), ...
```

where *a* and *b* were database calls, the user had to write them as

```
..., queryDB(a(A, B, C)), queryDB(b(F, A, R)), ...
```

This had the potential of retrieving the same information from *b* multiple times as multiple *a* tuples were extracted before a pair of solutions with a common *A* were found. However, this operation is just a join in standard RDBS terminology, and joins are something RDBSs do well.

During the second phase of interface development, a database compiler was written to allow the users to write programs with little thought about database systems. The compiler attempts to retrieve database information as efficiently as possible. The user places *db* declarations at the beginning of the application program, stating which predicates are database calls. For example, the user could say

```
:- db(parts/5), db(item/6)
```

to declare the two 5- and 6-place predicates *parts* and *item* as residing in the database. The program is then read in using *dbConsult(File)*, which reads *File* and asserts rewritten forms of the program clauses. This rewriting has two phases. The first simply scans the clause and replaces all calls having predicates declared with *db* by a *queryDB* applied to the call. Then, a pass over the rewritten clause optimizes the call. Currently, the only optimization attempted is that contiguous *queryDB*'s are merged into one *queryDB* call. For example,

```
a :- parts(A,B,C),item(A,F), F < 12000.
```

is changed to

```
a :- queryDB([parts(A,B,C),item(A,F)]), F < 12000.
```

At the moment, any other possible optimizations, such as passing range checks as shown above, are not performed. Also, disjunctions are ignored. As more optimizations are noticed, they will be added.

In the first phase of interface development, if a call such as *a(A,A)* was made, two items were passed back from DBMachine and unified together. This was fixed in the second phase. After sending information on each separate predicate to DBMachine, *requestDB* notices where variables are repeated and sends this information along. The restricted QUEL expression generated by the above *queryDB* would be

```
range of e0 is parts  
range of e1 is item  
  
retrieve (e0.col1,e0.col2,e0.col3,e1.col2)  
where e0.col1 = e1.col1
```

which is processed more quickly by Ingres than the unrestricted QUEL

```
range of e0 is parts  
range of e1 is item  
retrieve (e0.col1,e0.col2,e0.col3,e1.col1,e1.col2)
```

which has many more possible solutions. In one test, the query was a join on the first argument of *parts* and *item*, where *item* was a 6 place predicate with 20 tuples, and *parts* was a 5 place predicate with 14 tuples. Without the restriction, the cartesian product of 280 tuples was retrieved by Ingres, taking around 15 seconds. Approximately 10 seconds were then used to unify the possible combinations together until the correct tuple was found. With the restriction added, less than a second of processing time was needed. Variables with multiple occurrences in the query only elicit one value to be returned from DBMachine. For example, the above restricted join would return four items per tuple back to Prolog.

8.8.4. Future Research

Many problems remain to be solved. One problem is clogging of the atom table. When strings are returned from DBMachine, they are stored in Prolog's name table. With a large database, the name table will soon be clogged with new atoms from the database, even though many may be useless to the program. One possible solution to this problem is to have DBMachine assign a unique identifier to each atom Prolog hasn't already entered in the atom table. Also, more optimizations on the query can be performed, such as the range check mentioned above. Allowing variables and structures to reside in the database would be helpful. Another modification to improve efficiency would be to have DBMachine return Warren Abstract Machine code [Warren83]. This could then be executed by Prolog to retrieve the possible answers to the query.

The problems above only relate to retrieval from the database. In actual applications, the Prolog system will have to make updates to the database. An approach similar to [Warren84], with appropriate optimizations for bulk updates and the like, will be implemented.

8.8.5. Conclusion

Relational database systems and logic are so closely related, it seems that they must be joined to solve the problems of large-scale knowledge base management. There has been much debate as to the right approach to amalgamate the two. The interface system constructed in this project provides a flexible testbed in which to explore solutions to the difficult problems arising in this amalgamation. Work with this interface has indicated solutions to some questions such as the join problem. It is felt that the remaining problems can be similarly solved, resulting in a powerful amalgam of Prolog and relational database systems.

References

- Bowen, K.A., Buettner, K.A., Cicekli, I., and Turk, A., [1985]: *The Design and Implementation of a High-Speed Incremental Portable Prolog Compiler*, Tech Report CIS-85-4, Syracuse University. To appear in The International Logic Programming Conference '86 Proceedings.
- Chakravarthy, U.S., Minker, J., and Tran, D.: *Interfacing Predicate Logic Programs and Relational Databases*, University of Maryland, unpublished draft.
- Gallaire, H., and Minker, J. [1978]: *Logic and Databases*, Plenum Press, New York.
- Minker, J., [1978]: *An Experimental Data Base System Based on Logic*. In: *Logic and Databases* (eds. Gallaire, H. and Minker, J.), Plenum Press, New York, pp. 107-148.
- Reiter, R., [1978a]: *Deductive Question Answering on Relational Data Bases*. In: *Logic and Databases* (eds. Gallaire, H. and Minker, J.), Plenum Press, New York, pp. 149-176.
- Reiter, R., [1978b]: *Structuring a First-order Database*, Proceedings of the Canadian Society for Computational Studies of Intelligence.
- Stonebraker, M., Wong, E., Kreps, P., and Held, G. [1976]: *The Design and Implementation of Ingres*, ACM Transactions on Database Systems, 1:3, (September 1976).
- Warren, D.H.D., [1983]: *An Abstract Prolog Instruction Set*, SRI Technical Report.
- Warren, D.S., [1984]: *Database Updates in Pure Prolog*, Proceedings of the International Conference on Fifth Generation Computer Systems.

8.9 Hyung-Sik Park²

Negation and Databases

Hyung-Sik Park was a visiting assistant professor of Computer and Information Science during the academic year 1986-87, and worked on the grant during the spring term of 1988. (He accepted a position at the University of Iowa in June of 1988.) His research area is the interaction between logic deduction from databases and the assumption of the Generalized Closed World Assumption (GCWA) for those databases, which was the subject of his dissertation at Northwestern University under the direction of Prof. Lawrence J. Henschen. Under this approach, one separates the complete database available to the program into two distinct parts: The Extensional Database (EDB) consisting of ground atoms (facts) and the Intensional Database (IDB) consisting of all other available clauses. The assumption is that in large applications the size of the EDB will dwarf that of the IDB, and that typically, the EDB will be maintained on secondary storage while the IDB will often reside in main memory. The concern is that the solution of complex queries will lead to large volumes of retrievals of facts from the EDB. Since retrieval from the EDB will be measured in milliseconds as opposed to microsecond retrievals from the IDB, this would lead to serious inefficiencies in applications.

The approach taken here to this problem is based on the general compilation philosophy followed in the rest of the project: Determine at compile-time the EDB retrievals which can follow from use of a member of the IDB. Then at run time, various optimizations to speed availability of the EDB facts can be applied. This can range from semi-symbolic execution of the program - batching all retrievals to the end of symbolic execution, followed by retrieval and final resolution of the solution - to initiating parallel retrieval and caching of the EDB facts associated with an IDB rule the moment it is evident at run-time that the rule will be executed. For Horn clause databases, the basic theory of this approach has been worked out in [Chang,1981], [Henschen and Naqvi, 1984], and [Reiter, 1978a].

The treatment of negative information causes an increase in problems. Because of the potentially large volume of negative facts which must be stored if explicit representation were to be used, it is preferable to represent negative facts implicitly. This leads to the Closed World Assumption (CWA - [Reiter,1978b]): A ground fact is assumed to be false (i.e., its negation is true) if it cannot be deduced from the combination of the EDB and IDB. Otherwise stated, this is the *principle of negation by failure*: A negated ground atom is provable if the attempt to prove (via a complete positive deduction procedure) the unnegated ground atom fails. For Horn databases, the relation between negation by failure and logical negation is well-understood ([Clark, 1978]). However, for non-Horn IDBs, the CWA leads to contradictions. For example, if the IDB consists of $(p \vee q)$ alone, then neither p nor q is a logical consequence of the DB, so that under the CWA, both $\neg p$ and $\neg q$ would be provable, a contradiction. The difficulty arises because p (and also q) is *indefinite* with respect to the DB: neither p nor q is a logical consequence of the DB. The example demonstrates that the negation by failure approach does not distinguish between genuinely false atoms (relative to the DB) and those which are indefinite relative to the DB.³

Define PIGC to be the set of minimal positive ground clauses implied by the DB, where a clause is minimal if it is not properly subsumed by any positive clause deducible from the DB. For a ground atom q , $\text{PIGC}[q]$ consists of those elements of PIGC in which q occurs positively as a subformula. The Generalized Closed World Assumption states that if q is a ground atom, then $\neg q$ can be assumed true if q is not deducible from DB and q is not indefinite with respect to DB. Let

²This section written by K.A. Bowen.

³The unpleasant nature of DBs which leave some formulas indefinite is long-established: Classical logic and model theory - e.g., modern proofs of the Completeness Theorem - extensively utilize methods (Lindenbaum's Lemma) which embed initial consistent theories or DBs in complete extensions; i.e., in supersets which leave no formulas indefinite. Unfortunately, although Lindenbaum's Lemma guarantees the existence of complete extensions (assuming the Axiom of Choice), the problem of obtaining such extensions is recursively unsolvable. Consequently, even in those settings where passing to a complete extension would be logically reasonable, it is not computationally possible. Hence the need to compute the set of indefinite formulas of the original theory. -KAB

us write $GCWA(DB, -q)$ for this state of affairs. It follows from [Minker, 1982] that $GCWA(DB, -q)$ if and only if $PIGC[q]$ is empty. It follows that the problem of coping with indefinite formulae can be reduced from treating the entire set of indefinite formulae (with respect to DB) can be reduced to computing the indefinite formulae relevant to the query at hand.

Further reductions are possible. The following representations are known (Henschen and Park, [Henschen and Park, 1986]):

(1) $PIGC[q]$ with respect to DB is equivalent to $PIGC[q]$ with respect to $EDB \cup NUGF \cup NH[q] \cup PSUB[nhi]$, where

$$CDB = IDB \cup NNUC,$$

(2) $PIGC[q]$ with respect to DB is equivalent to $PIGC[q]$ with respect to $EDB \cup NH[q] \cup PSUB[nhi]$, where

$$CDB = IDB \cup NC,$$

In both cases, $NH[q]$ is the set of minimal non-Horn clauses containing a positive occurrence of the predicate of q and are derivable from CDB , $NNUC$ is the set of negative nonunit clauses, and $NUGF$ is the set of negative unit ground facts, NC is the set of negative clauses, and $PSUB[nhi]$ is the set of clauses derivable from CDB and which potentially subsume some clause in $NH[q]$.

All of the foregoing results are valid for DB s whose formulae contain no function symbols. During the spring term, Park investigated methods for possible extension of the results to settings in which function may be present, as well as possible further improvements of the reductions and resulting computations of $GCWA[q, DB]$. Several suggestive special cases appeared, but as yet no general conclusions can be drawn.

Park also organized and conducted a research seminar on Expert Database Systems for the staff of the grant, as well as other graduate students in the department. The outline of the topics of the seminar was:

- Introduction of expert systems, databases, and expert database systems.
- Knowledge-based systems, knowledge representation, logical analysis of knowledbases, incompleteness, commonsense reasoning, non-monotonicity, and reason-maintenance systems.
- Database management systems, semantic data modelling, database constraints, dependencies, and normal forms, dextensions of DBMSs, including deductive databases, incomplete databases, and temporal databases.
- Knowledge base management systems and architectures, logic-based data languages, recursion, complex objects, object-oriented paradigms in KBMSs, constraint management, semantic query optimization, knowledge engineering in DBMSs, intelligent KB-interfaces.

References

- [Chang] Chang, C.L., *On evaluation of queries containing derived relations*, in **Advances in Database Theory**, vol. 1, H.Gallaire, J.Minker, and J.M. Nicolas, eds, Plenum Press, New York, 1981, 235-260.
- [Clark] Clark, K.L., *Negation as failure*, in **Logic and Databases**, H.Gallaire and J.Minker, eds, Plenum Press, New York, 1978, 293-324.
- [Henschen and Naqvi], Henschen, L.J. and Naqvi, S., *On compiling queries in recursive first-order databases*, **JACM**, 31:1(1984), 47-85.

[Henschen and Park] Henschen, L. and Park, H-S., *Indefinite and GCWA inference in indefinite deductive databases*, in **Proc. AAAI National Conference**, 1986.

[Minker] Minker, J., *On indefinite databases and the closed world assumption*, in **Lecture Notes in Computer Science**, 138, Springer Verlag, 1982, 292-308.

[Reiter, 1978a] Reiter, R., *Deductive question answering on relational databases*, in **Logic and Databases**, H. Gallaire and J. Minker, eds, Plenum Press, New York, 1978, 149-177.

[Reiter, 1978b] Reiter, R., *On closed world databases*, in **Logic and Databases**, H. Gallaire and J. Minker, eds, Plenum Press, New York, 1978, 55-76.

Appendix Published Papers⁴

Hamid Bacha, *Meta-level Programming: A Compiled Approach*, **Proceedings of the Fourth International Conference on Logic Programming**, Edited by Jean-Louis Lassez, MIT Press, 1987.

Howard A. Blair and V.S.Subrahmanian, *Paraconsistent Logic Programming*, in **Proc. 7th Intl. Conf. on Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science**, 287, (1987) 340-360. Invited for resubmission to **Theoretical Computer Science**.

⁴Upwards of 10 additional papers by grant personnel either have recently been submitted for publication, or will shortly be completed and submitted during the spring of 1988.